Пояснювальна записка
до дипломного проекту

на тему: _____*Цифровий ключ для транспортного засобу*_____
_____
_____

Київ – 2019

ЗМІСТ

СЛОВНИК ВИКОРИСТАНИХ СКОРОЧЕНЬ ТА ТЕРМІНІВ 5
ВСТУП 11
1 ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ 12
2 ТЕХНІЧНІ ХАРАКТЕРИСТИКИ 13
2.1 Опис системи. 13
2.2 Вимоги до системи 14
2.2.1 Вимоги до серверної частини системи 14
2.2.2 Вимоги до мобільного додатку 15
2.2.3 Вимоги до модуля транспортного засобу 15
3 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ 16
3.1 Мобільний додаток **Key Programming Procedure** 16
3.2 Мобільний додаток **SLICK** 17
3.3 **Car Connectivity Consortium Digital Key** 18
3.4 Висновки 19
4 РОЗРОБКА ЗАСОБІВ ТА ІНСТРУМЕНТІВ ДЛЯ РОБОТИ З СЕРВЕРНИМИ КОМПОНЕНТАМИ 20
4.1 Розробка інструментів для роботи з сервісом аутентифікації 20
4.2 Розробка інструментів для роботи з БД 21
4.2.1 Опис структури БД 21
4.2.2 Опис засобів роботи з БД 26
4.3 Опис засобів роботи зі сховищем медіа даних 27
4.4 Висновки 28
5 НАЛАШТУВАННЯ СЕРВЕРНИХ КОМПОНЕНТІВ 29
5.1 Створення Firebase проекту 29
5.2 Налаштування сервісу аутентифікації 31
5.3 Налаштування БД 32
5.4 Налаштування сховища медіа даних 33

5.5 Розробка та розгортання додаткових серверних функцій 34
5.5.1 Опис Firebase Cloud Functions 34
5.5.2 Розробка і розгортання Firebase Cloud Functions 34
5.6 Висновки 37
6 РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ 38
6.1 Розробка активності аутентифікації 38
6.1.1 Розробка екрану реєстрації 38
6.1.2 Розробка екрану підтвердження електронної пошти 39
6.1.3 Розробка екрану логіну 40
6.2 Розробка активності верифікації власника транпортного засобу 41
6.2.1 Розробка екрану додавання транспортного засобу 42
6.3 Розробка основної активності 43
6.3.1 Розробка навігаційної панелі 43
6.3.2 Розробка екрану транспортних засобів 44
6.3.3 Розробка екрану деталей транспортного засобу 45
6.3.4 Розробка екрану шерінгу цифрового ключа 46
6.3.5 Розробка екрану друзів 47
6.3.6 Розробка екрану дефолтних властивостей цифрового ключа 49
6.3.7 Розробка екрану цифрових ключів 50
6.3.8 Розробка екрану налаштувань 52
6.4 Розробка BLE сервісу 53
6.5 Висновки 54
7 РОЗРОБКА ДОДАТКУ ТРАНСПОРТНОГО ЗАСОБУ 55
7.1 Налаштування модуля додатку 55
7.2 Розробка головної активності 56
7.3 Розробка контроллера керування замком транспортного засобу 56
7.4 Розробка компонентів керування Bluetooth-ом 57
7.4.1 Сервіс BLE 57
7.4.2 Задача пошуку QKey девайсів 58

7.4.3 Засоби оцінки відстані між BLE пристроями 58
7.4.4 VehicleGattServerCallback..... 59
7.5 Розробка менеджера цифрових ключів..... 60
7.6 Висновки 61
8 РОЗРОБКА ПРОТОТИПУ 62
8.1 Налаштування одноплатного мікрокомп’ютера 62
8.2 Імітація замка транспортного засобу 67
8.2.1 Сервопривод Tower Pro SG90 67
8.2.2 Розробка контроллера для сервоприводу 68
8.3 Проектування прототипу транспортного засобу 70
8.4 Висновки 71
ВИСНОВКИ..... 72
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 75
Додаток А 76
Додаток Б..... 82
Додаток В 86
Додаток Г..... 92

СЛОВНИК ВИКОРИСТАНИХ СКОРОЧЕНЬ ТА ТЕРМІНІВ

Англомовний варіант	**Україномовний варіант**	**Опис**
10/100 Ethernet		
3.5mm Analog Output	3.5 мм Аналоговий Вихід	Аудіо роз'єм
CCC (Car Connectivity Consortium)		Міжнародна організація
Activity	Активність	Один з найважливіших компонентів Android додатку
Android	Андроід	Мобільна операційна система
Android Studio	Андроід Студіо	Середовище розробки Android додатків
Android Things	Андроід Речей	Операційна система
Android Things Console		Додаток управління продуктами для ОС Android Things
Android Things Setup Utility		
API (Application Programming Interface)	Інтерфейс Програмування Додатків	
ARM Cortex A53		Процессор
AsyncTask		Клас Android SDK для виконання роботи у фоновому режимі
AuthManager		Інтерфейс для роботи із сервісом аутентифікації Firebase
AuthManagerImpl		
Backend	Бекенд	Серверна частина ПЗ
BaseHelper		Базовий інтерфейс для роботи Firebase RTDB
BLE (Bluetooth Low Energy)		Різновид технології Bluetooth 4.+
BluetoothAdapter		
BluetoothManager		

Broadcom BCM2837		ОЗП
Child	Нащадок	Молодший node у Firebase RTDB
CPU (Central Processing Unit)	ЦП (Центральний процессор)	
DatabaseReference	Посилання БД	Посилання на певний child Firebase RTDB
DB (Database)	БД (База даних)	
DbHelper		Інтерфейс-фасад для роботи з усією БД Firebase RTDB
DbScheme		Файл, який містить опис структури БД проекту
Digital Key	Цифровий ключ	
DigitalKeyManager	Менеджер цифрових ключів	
ER (Entity-Relationship)	СЗ (Сутність-Зв'язок)	
Firebase Authentication		Сервіс аутентифікації Firebase
Firebase CLI		Firebase консоль
Firebase Cloud Functions		Сервіс розширення функціоналу Firebase
Firebase Management Console	Консоль управління Firebase	
Firebase Realtime Database, RTDB		NoSQL Firebase БД
Firebase Storage		Сховище медіа даних Firebase
Firebase tools	Інструменти Firebase	
FirebaseAuth		Основний клас Firebase SDK для роботи із сервісом аутентифікації
Fragment	Фрагмент	UI компонент Android SDK
GATT (Global Attribute)		Протокол BLE
GPIO (Global Platform Input Output)		Інтерфейс для зв'язку між компонентами комп'ютерної системи
GPS (Global Positioning System)	СГП (Система глобального позиціонування)	

GPU (Graphic Processing Unit)	ГП (Графічний процессор)	
Gradle		Система автоматичної збірки
HDMI (High Definition Multimedia Interface)		
I2C (Inter-Integrated Circuit)		Послідовна асиметрична шина для зв'язку між інтегральними схемами
Index js		Файл, який містить код функції Firebase Cloud Function
IoT (Internet of Things)	Інтернет речей	
Java		Мова програмування
JavaScript		Мова програмування
JSON (JavaScript Object Notation)		Формат зберігання та передавання даних
Jumper		Різновид дротів
Key	Ключ	Сутність системи QKey
Key Programming Procedure		Назва мобільного додатку
Key properties functions		Firebase Cloud Functions, пов'язані з сутністю KeyProperties
KeyHelper		Допоміжний клас для роботи з сутністю Key Firebase RTDB
KeyProperties	Властивості ключа	Сутність системи QKey
KeyPropertiesHelper		Допоміжний клас для роботи з сутністю KeyProperties Firebase RTDB
Kotlin		Мова програмування
LockController	Контроллер замка	
Logger	Реєстратор	
Male-female		Тип дротів різновиду Jumper
Measured power	Вимірювана потужність	Потужність сигналу Bluetooth, яка очікується на

		відстані 1 м між пристроями
MediaHelper		Інтерфейс для роботи зі сховищем медіа даних
MicroSD		Різновид карток пам'яті
Navigation Architecture Components	Навігаційні архітектурні компоненти	Компонент Android SDK
NavigationDrawer		Компонент Android SDK
NFC (Near Field Cummunication)	Ближній безконтактний зв'язок	Технологія передачі даних
Node	Вузол	Вузол у структурі RTDB
Node js		Платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків
NoSQL		БД, яка має не SQL структуру
OS (Operating System)	ОС (Операційна Система)	
Parent	Пращур	Старший node у Firebase RTDB
Play Services		Сервіси Android SDK від компанії Google
POLICY_CUSTOM		Різновид політики цифрового ключа системи QKey
PWM (Pulse Width Modulation)	ШІМ (Широтно-імпульсна модуляція)	
PWM0		Назва GPIO піну RPi
QKey		Назва додатків проекту
QKeyDigitalKeyManager		Менеджер цифрових ключів додатку транспортного засобу
RAM (Random Access Memory)	ОЗП (Оперативний запам'ятовуючий пристрій)	
Raspberry Pi 3 Model B, RPi		IoT девайс

REST (Representational State Transfer)	Передача стану представлення	Архітектурний стиль
RPi Camera module v2		RPi інтерфейс камери
SDK (Software Development Kit)	Набір засобів розробки	
ServoLockController		Контроллер замка-імітатора на основі сервоприводу
SharedPreferencesManager		Менеджер приватного сховища даних типу "ключ-значення" додатків QKey
ShareGroup		Сутність системи QKey
ShareGroupHelper		Допоміжний клас для роботи з сутністю ShareGroup Firebase RTDB
SLICK		Мобільний додаток
SPI (Serial Peripheral Interface)	Послідовний периферійний інтерфейс	Послідовний синхронний стандарт передачі даних в режимі повного дуплексу
SQL (Select Query Language)		Мова запитів до БД
StorageHelper		Інтерфейс для роботи зі сховищем медіа даних
StorageReference		Посилання на репозиторій Firebase Storage
TextViewLogger		Реалізація Logger
Tower Pro SG90		Сервопривод
TypeScript		Мова програмування
UART (Universal Asynchronous Receiver-Transmitter)	УАПП (Універсальний асинхронний прийомопередавач)	
UI (User Interface)	Користувацький інтерфейс	
UID (Unique Identifier)	Унікальний ідентифікатор	
UML (Unified Modeling Language)	Уніфікована мова моделювання	
Ungrouped	Немає групи	
URL (Uniform Resource Locator)	Єдиний Локатор Ресурсу	

USB 2.0 (Universal Serial Bus)	Універсальна послідовна шина	
User	Користувач	
User-space driver		Різновид периферійних драйверів ОС Android Things
UserHelper		
Vehicle	Транспортний засіб	
VehicleBleService		BLE слежба додатку QKey Vehicle
VehicleGattServerCallback		Клас додатку QKey Vehicle для роботи з GATT сервером додатку QKey
VehicleHelper		Допоміжний клас для роботи з сутністю Vehicle Firebase RTDB
VehicleUtils		Vehicle специфічний допоміжний клас
VideoCore IV		ГП

ВСТУП

На сьогоднішній день, різноманітні види транспорту заполонили увесь світ. Також нормою і буденністю стає стрімке розповсюдження смартфонів, розумних мобільних телефонів, які вже сьогодні задовольняють потреби більшості користувачів, вирішуючи найрізноманітніші повсякденні задачі своїх власників. Зважаючи на те, що скоріш за все кожен водій будь-якого транспортного засобу володіє смартфоном, з’явилась ідея дати змогу власникам смартфонів використовувати їхні девайси замість реального ключа. Реалізація цієї ідеї сприятиме розвиту обох ринків, транспорту та мобільного зв’язку, та багатьох інших. За вдалої реалізації очевидне стрімке зростання популярності каршерінгових сервісів.

Наразі, основною проблемою для створення такого продукту є надто широкий асортимент транспортних засобів та недоліки додатків, які існують на ринку і підривають довіру користувачів.

Таким чином, для досягнення дійсно гарного результату, потрібно розробити систему, яка складалася б з універсальних, надійних та привабливих для ринку компонентів з економічної та ергономічної точок зору.

Результати даної роботи спрямовані на розробку демонстраційного прототипу, який використовуватиметься для подальшого розвитку концепції.

1 ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Призначення даної системи полягає у спрощенні використання транспортного засобу, адже смартфон, як правило, людина завжди носить при собі. Також збільшується захист транспортного засобу за рахунок внутрішніх захисних механізмів мобільного додатку. Однією з основних перспективних функцій системи є функція шерінгу цифрового ключа, тобто надання прав користування ключем іншій особі. За рахунок цього, власник транспорту має змогу чітко і просто регулювати доступ до ключів, а також надати змогу користування іншій людині навіть за відсутності змоги передати людині реальний фізичний ключ. В додачу, власник отримує змогу накладати різноманітні обмеження на користування: на час, дату, гео-положення транспортного засобу, а також рівень доступу. Не менш важливою є функція ведення статистики використання засобу. За рахунок інформатизації з’являється можливість записування даних про користування транспортним засобом та подання цієї інформації у зручному для користувача вигляді.

Області застосування в широкому сенсі поділяються на дві великі категорії: для власного користування і для бізнесу. Чітко споглядається перспектива використання системи у каршерінгових сервісах і різноманітних пасажироперевізних компаніях, таких як служби таксі, або й навіть постачальники послуг громадського транспорту.

2 ТЕХНІЧНІ ХАРАКТЕРИСТИКИ

2.1 Опис системи.

Вихідна система представлена на Рисунку 2.1. Детальніше про компоненти системи можна дізнатися за допомогою Діаграми компонентів.

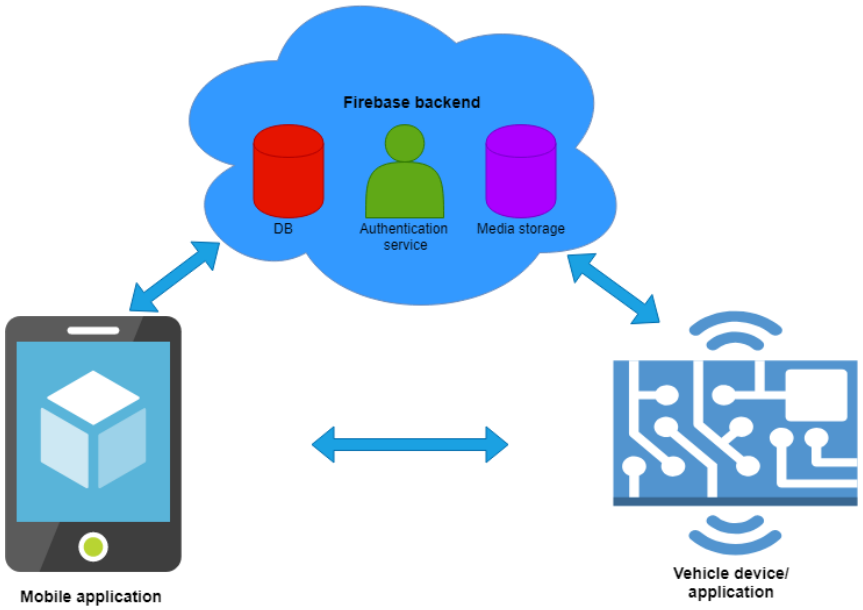


Рисунок 2.1 – Система цифрових ключів для транспортних засобів

Розглянемо принцип роботи даної системи.

Основний, центральний компонент системи являє собою backend, який функціонує як сервіс що використовується в якості серверної частини системи. Розроблений компанією **Firebase**, дочірньою компанією конгломерату **Google**. Цей модуль надає засоби для використання наступних надважливих сервісів і служб:

- Бази даних – **Firebase Realtime Database**;
- Сервісу аутентифікації – **Firebase Authentication**;
- Сховища медіа даних – **Firebase Storage**.

Також для реалізації певного функціоналу використовується сервіс **Firebase Cloud Functions**, який дозволяє розширити базові можливості Firebase.

Наступний компонент системи – мобільний додаток. В рамках дипломного проекту, під мобільним додатком мається на увазі мобільний додаток для найпоширенішої мобільної операційної системи **Android**, створеної компанією Google.

Останній компонент системи – модуль транспортного засобу, який складається з IoT девайсу і додатку, який надає функціонал користувачеві та керує необхідними системами транспорту. В рамках дипломного проекту, не ставиться за мету створити модуль, який інтегруватиметься у певний транспортний засіб. Навпаки мета проекту – створити демонстраційний модуль для прототипу, який використовуватиме універсальні інтерфейси інтеграції. В свою чергу, даний модуль складається з 3 основних компонентів: власне IoT девайсу, MicroSD картки пам'яті для зберігання образу ОС та додатку і сервоприводу.

2.2 Вимоги до системи

Як і було описано раніше, мета і головне завдання на даному етапі – створити прототип. Тому вимоги до системи під час виконання дипломного проекту фільтруються крізь загальний план створенная продукту.

2.2.1 Вимоги до серверної частини системи

Наразі, необхідно надати користувачеві можливість зберігати власні дані та медіа файли, а також забезпечити аутентифікацію користувачів за допомогою електронної пошти та паролю.

Аутентифікаційний функціонал включає в себе: реєстрацію та верифікацію користувачів, а також можливість зміни пароля в разі його втрати. В процесі реєстрації, поштова адреса користувача обов'язково має бути верифікована.

Також необхідно розширити базовий функціонал бекенду функцією автоматичного відправлення електронних листів із системною інформацією.

2.2.2 Вимоги до мобільного додатку

Операційна система: **OS Android**.

Мінімальна пітдримувана версія OS Android: **5.0**.

Мова програмування додатку: **Kotlin, Java**.

Основний функціонал, який має бути реалізований під час виконання дипломного проекту: усі аутентифікаційні можливості користувача, функція додавання транспортного засобу, функція шерінгу цифрового ключа та функції оперування цифровим ключем для взаємодії із транспортом.

2.2.3 Вимоги до модуля транспортного засобу

Операційна система IoT додатку: **Android Things**.

Мова програмування додатку: **Kotlin, Java**.

IoT девайс для прототипу: **Raspberry Pi 3 Model B**.

Підтримувані технології комунікації із моб. додатком: **Bluetooth/BLE**. Пізніше планується реалізація комунікації за технологією **NFC** в якості превентивного методу передачі даних.

В якості прототипного замка транспортного засобу використовується **сервопривод**.

3 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

3.1 Мобільний додаток **Key Programming Procedure**

Ідея цифрового ключа для транспортного засобу, звичайно, не нова. Існує багато рішень, які тою чи іншою мірою реалізують необхідний функціонал. Поштовх для розвитку цієї ідеї надала технологія безконтактної передачі даних NFC, яка швидко знайшла собі застосування завдяки розвитку смартфонів.

Найперша спроба замінити реальний фізичний ключ цифровим була зроблена на рівні мобільного додатку виходячи з твердження про те, що за допомогою декількох нескладних кроків можна імітувати реальний ключ у додатку.

Одним з таких додатків є Key Programming Procedure, який імітує фізичні ключі різних брендів та моделей авто.

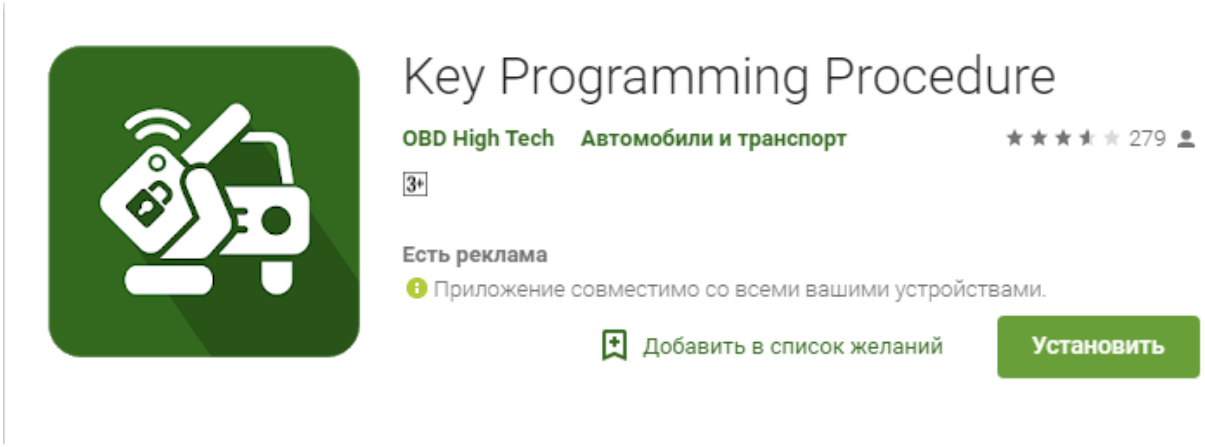


Рисунок 3.1 – Додаток Key Programming Procedure

На сьогоднішній день цей продукт є найгіршим та не завойував прихильності і довіри користувачів.

Переваги:

- Простота встановлення додатку.

Недоліки:

- Не продумана архітектура;

- Надто вузький спектр використання (тільки автомобілі, лише певних моделей);
- Відсутність продуманого, надійного захисту;
- Недосконала реалізація системи.

3.2 Мобільний додаток **SLICK**

Гарний приклад реалізації технології цифрового ключа – мобільний додаток SLICK. На відміну від Key Programming Procedure, SLICK являє собою цілковиту систему, яка не імітує, а реалізує функції цифрового ключа.

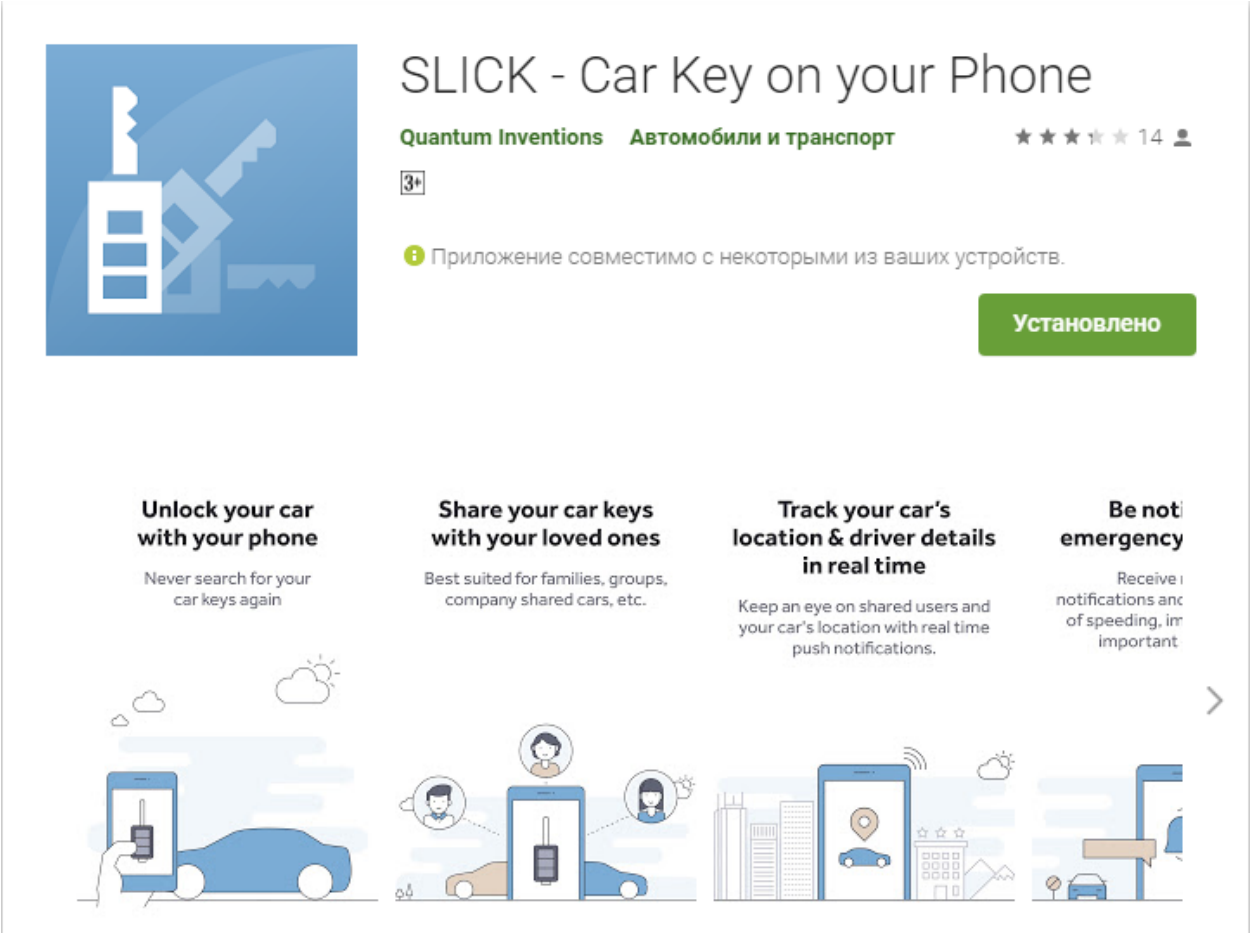


Рисунок 3.2 – Додаток SLICK

Переваги:

- Широкий спектр запропонованих функцій;
- Продуманий, привабливий мобільний додаток.

Недоліки:

- Складна процедура інтеграції із автомобілем;
- Вузький спектр використання (тільки автомобілі).

3.3 **Car Connectivity Consortium Digital Key**

Ідею цифрового ключа розвинула міжнародна організація **Car Connectivity Consortium**.

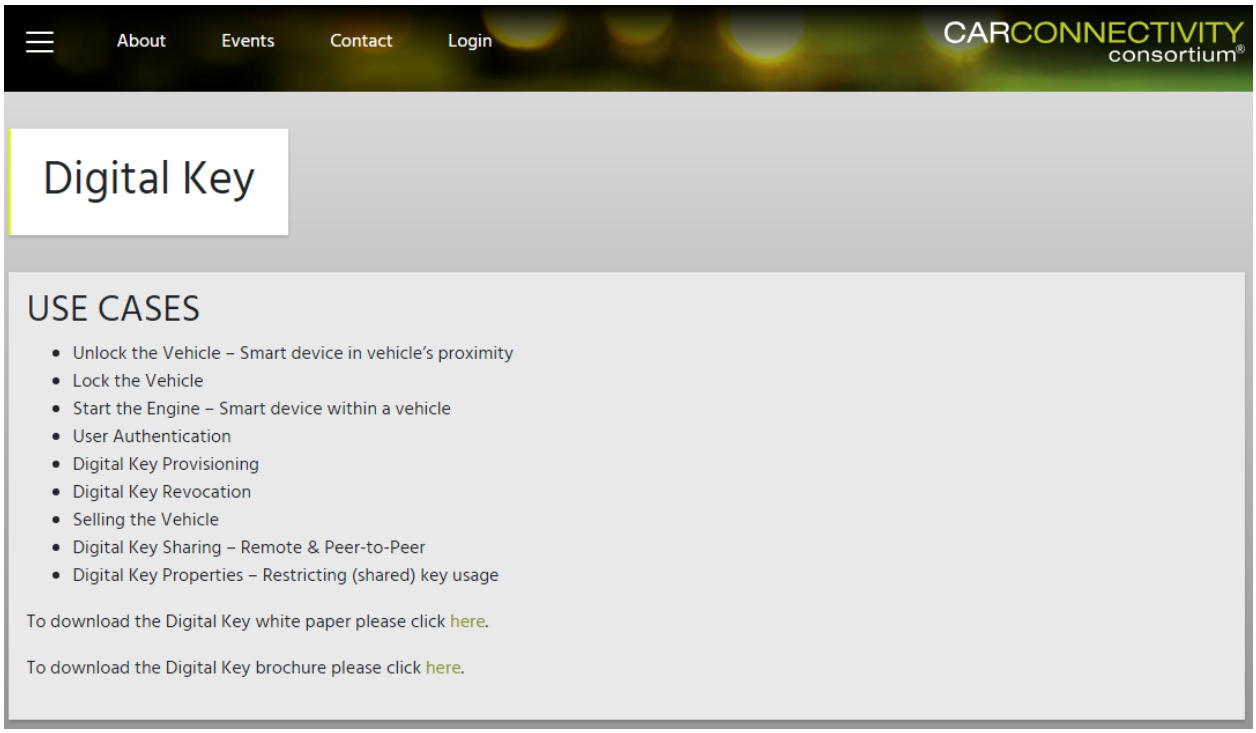


Рисунок 3.3 – Опис технології Digital Key від Car Connectivity Consortium

Наразі цей продукт є найбільш досконалим з ідейної точки зору.

Переваги:

- Продумана архітектура системи;
- Велика кількість партнерів-провайдерів авторинку.

Недоліки:

- Відсутність конкретної реалізації мобільного додатку;
- Відсутність конкретної реалізації автомобільної системи;

- Вузький спектр використання (тільки автомобілі).

3.4 Висновки

В ході аналізу даної предметної області та огляду існуючих рішень було виявлено позитивні та негативні моменти кожного рішення, обрано оптимальний шлях розробки власної системи цифрових ключів для транспортних засобів.

4 РОЗРОБКА ЗАСОБІВ ТА ІНСТРУМЕНТІВ ДЛЯ РОБОТИ З СЕРВЕРНИМИ КОМПОНЕНТАМИ

4.1 Розробка інструментів для роботи з сервісом аутентифікації

Firebase забезпечує декілька шляхів використання власних сервісів: REST API або SDK. Звісно ж використовувати SDK для доступу до серверних служб набагато легше, адже більшість рутинної роботи вже була зроблена розробниками і з'являється можливість сконцентруватися на більш важливих питаннях під час проектування системи.

Задля забезпечення універсальності розроблюваних інструментів, створюємо окремий Gradle модуль в Android проекті. **Gradle** – система збірки, компонування файлів проекту у вихідний файл, яка використовується у Android Studio.

Надалі починається власне створенная засобів роботи із службою аутентифікації Firebase Authentication. З огляду на те, який саме функціонал потрібен для даного проекту, вирішено було створити єдиний інтерфейс AuthManager, який надаватиме увесь спектр необхідних функцій клієнту, тобто додаткам системи. Отже інтерфейс AuthManager дає клієнтові наступний ряд можливостей: реєстрація та логін за електронною поштою і паролем, вихід з програми, перевірка стану користувача всередині системи (верифікація поштової адреси, виконання процедури входу/виходу), надання важливих даних про юзера, а також методи для відсилання системних електронних повідомлень на адресу користувача (верифікація поштової адреси, зміна паролю). Слід також відмитити, що стан користувача не змінюється після термінації додатку, адже він зберігається серед кешованих даних Firebase Authentication SDK.

Для реалізації інтерфейсу, було створено клас AuthManagerImpl, який використовує інструмент Firebase SDK під назвою FirebaseAuth. Цей клас надає змогу легко і швидко імплементувати увесь необхідний функціонал, оминаючи складнощі і тонкощі роботи серверу.

Таким чином було розроблено API для клієнтської частини системи, яке приховує деталі реалізації та роботи бекенду, натомість надаючи змогу додаткам централізовано керувати усіма аутентифікаційними процесами.

4.2 Розробка інструментів для роботи з БД

Для роботи із базою даних, необхідно спочатку розробити структуру БД, яка охопить усю предметну область даного проекту. На відміну від багатьох інших сучасних баз даних, Firebase Realtime Database – це NoSQL база даних і це обумовлює її специфіку. По-перше, дані у Realtime Database зберігаються відмінним від SQL БД чином – у вигляді великого структурованого JSON файлу замість таблиць. По-друге, методи доступу та опрацювання даних також відрізняються, що зумовлено структурними особливостями RTDB. Основні поняття, якими оперує RTDB – це node-и, які в свою чергу поділяються на parent-ів та child-ів. В термінах JSON фоормату, parent-и і child-и поділяються між собою лише ієрархією ключів всередині файлу.

4.2.1 Опис структури БД

Загальний вигляд структури БД можна побачити на UML ER-діаграмі, зображеній схематично.

Отже, на діаграмі представлено п’ять сутностей, кожна з яких володіє рядом атрибутів. Почнемо з сутності **User.** Вона втілює найважливішу сутність даної предметної області – користувача. Звичайно що User займає центральне місце у даній схемі. Наразі він наділений наступними атрибутами:

- **id** – унікальний ідентифікатор кожного юзера, еквівалентний UID ідентифікатору, який автоматично генерується Authentication сервісом під час реєстрації користувача у системі;
- **username** – унікальний ідентифікатор користувача, який використовується власне юзерами для ідентифікації один одного у мобільному додатку, може бути змінений користувачем, за

замовчуванням він еквівалентний атрибуту id, за одним винятком – на початку додано символ @;

- **firstname** – ім’я користувача;
- **lastname** – прізвище користувача;
- **userId** – реальний ідентифікатор користувача, який використовується для його верифікації під час додавання транспортного засобу, пізніше планується релізувати можливість автоматичної перевірки документів юзера, які використовуватимуться в якості userId атрибуту;
- **photoUrl** – URL посилання на місце зберігання аватару користувача у Firebase Storage;
- **defaultKeyPropertiesId** – унікальний ідентифікатор дефолтного значення KeyProperties для даного користувача, створюється автоматично після реєстрації користувача у системі;
- **keys** – масив унікальних ідентифікаторів цифрових ключів, які асоційовані із цим юзером.

Сутність User має зв’язок із сутністю Key один-до-багатьох, а також один-до-одного із сутністю KeyProperties та один-до-багатьох із сутністю Vehicle. Запис до бази даних відбувається під час реєстрації, після підтвердження адреси електронної пошти користувачем.

Наступна сутність – **Vehicle**, яка представляє транспортний засіб. Наразі передбачається, що запис даних про транспортні засоби у БД відбувається на стороні транспортного провайдера задля забезпечення захисту.

Об’єкти Vehicle наділені наступними атрибутами:

- **id** – унікальний ідентифікатор кожного Vehicle запису, генерується провайдером транспорту під час реєстрації у системі;
- **ownerId** – ідентифікатор юзера, який володіє даним авто, наразі передбачається, що юзер використовуватиме паспортні дані у якості ідентифікатора;
- **name** – назва транспортного засобу;

- **vehicleId** – ідентифікатор транспорту, наразі передбачається, що це номер транспортного засобу, за яким його зареєстровано;
- **lastLocation** – опис останнього місцезнаходження транспортного засобу;
- **photoUrl** – URL посилання на місце зберігання зображення транспортного засобу у Firebase Storage;
- **defaultKeyPropertiesId** – унікальний ідентифікатор дефолтного значення KeyProperties для даного транспортного засобу, створюється автоматично після додавання даного транспорту до свого списку користувачем;
- **keys** – масив унікальних ідентифікаторів цифрових ключів, які асоційовані із цим транспортним засобом.

Сутність Vehicle має зв'язок із сутністю Key один-до-багатьох, а також один-до-одного із сутністю KeyProperties та багато-до-одного із сутністю User.

Наступна сутність – **Key**, яка представляє цифровий ключ. Ця сутність має найголовніше значення для роботи системи, адже саме вона поєднує усі сутності у цілісну структуру, придатну до використання системою.

Об'єкти Key наділені наступними атрибутами:

- **id** – унікальний ідентифікатор кожного Key запису, генерується автоматично під час створенная запису;
- **userId** – унікальний ідентифікатор юзера у системі, якому надано доступ користування транспортним засобом;
- **ownerId** – унікальний ідентифікатор юзера у системі, який володіє транспортним засобом, userId еквівалентий ownerId у випадку, коли цифровий ключ створено для власника транспорту;
- **vehicleId** – унікальний ідентифікатор транспортного засобу, для якого створено цифровий ключ;
- **policy** – атрибут, який визначає політику користування даним ключем, саме від нього залежить, які властивості ключа будуть враховуватися під час користування транспортним засобом;
- **status** – атрибут, що визначає статус активації ключа;

- **groupId** – унікальний ідентифікатор групи, до якої належить цей ключ, може бути відсутній в разі, якщо власник транспортного засобу не додав юзера, якому надав даний ключ до жодної групи;
- **keyPropertiesId** – унікальний ідентифікатор дефолтного значення KeyProperties для даного ключа, створюється автоматично після його створення.

Сутність KeyProperties має багато зв'язків. Вона пов'язана із сутністями User, Vehicle та ShareGroup зв'язками один-до-багатьох, а також зв'язком один-до-одного із сутністю KeyProperties.

Наступна сутність – **ShareGroup**, яка представляє групу користувачів. Ця сутність створюється власником транспортного засобу для об'єднання певних користувачів, яким він надав доступ до власного транспорту. Основне призначення групи – колективне, централізоване налаштування обмежень користування транспортом для певної групи людей, наприклад сім'ї або колег.

Об'єкти ShareGroup наділені наступними атрибутами:

- **id** – унікальний ідентифікатор кожного ShareGroup запису, генерується автоматично під час створенная запису;
- **ownerId** – ідентифікатор юзера, який створив дану групу;
- **name** – назва групи;
- **defaultKeyPropertiesId** – унікальний ідентифікатор дефолтного значення KeyProperties для даної групи, створюється автоматично після створення групи;
- **keys** – масив унікальних ідентифікаторів цифрових ключів, які асоційовані із цією групою.

Сутність ShareGroup пов'язана зв'язком багато-до-одного із сутністю User, а також зв'язками один-до-багатьох із сутністю Key та один-до-одного із сутністю KeyProperties.

Наступна і остання сутність – **KeyProperties**, об’єкти якої являють собою властивості певного цифрового ключа або ж навіть цілої групи ключів, якщо це властивості за замовчуванням для іншого об’єкта.

Об’єкти сутності KeyProperties наділені наступними атрибутами:

- **id** – унікальний ідентифікатор кожного KeyProperties, генерується автоматично під час створенная запису;
- **accessRestrictions** – об’єкт, який описує обмеження з користування цифровим ключем на рівні доступу, можливі наступні рівні доступу: *“Повний доступ”, “Відкрити/закрити (повний доступ)”, “ Відкрити/закрити (тільки багажник)”, “Запуск двигуна”*;
- **geoRestrictions** – об’єкт, який описує обмеження з користування цифровим ключем на рівні гео-положення транспортного засобу, реалізація даного обмеження не розглядається в рамках дипломного проекту;
- **dateTimeResctrictions** – об’єкт, який описує обмеження користування цифровим ключем в часі і даті;
- **usageCountRestrictions** – об’єкт, який описує обмеження користування цифровим ключем по кількості.

Будь-які обмеження можуть бути відсутні, навіть усі разом, в такому випадку користування цифровим ключем буде не обмежене жодним чином.

Сутність KeyProperties пов’язана з усіма іншими зв’язками один-до-одного.

Для розміщення даних у Firebase Realtime Database було обрано просту, але ефективну структуру: об’єкти усіх описаних вище сутностей, зберігаються кожний у власному child node-і, які мають відповідну назву та знаходяться в корені БД, далі – parent-и. Для розмежування записів та простої навігації базою, кожен об’єкт розташовується у власному child-і, ім’я якого (ключ) співпадає із унікальним ідентифікатором об’єкта. Приклад цієї структури зображено на Рисунку 4.2.

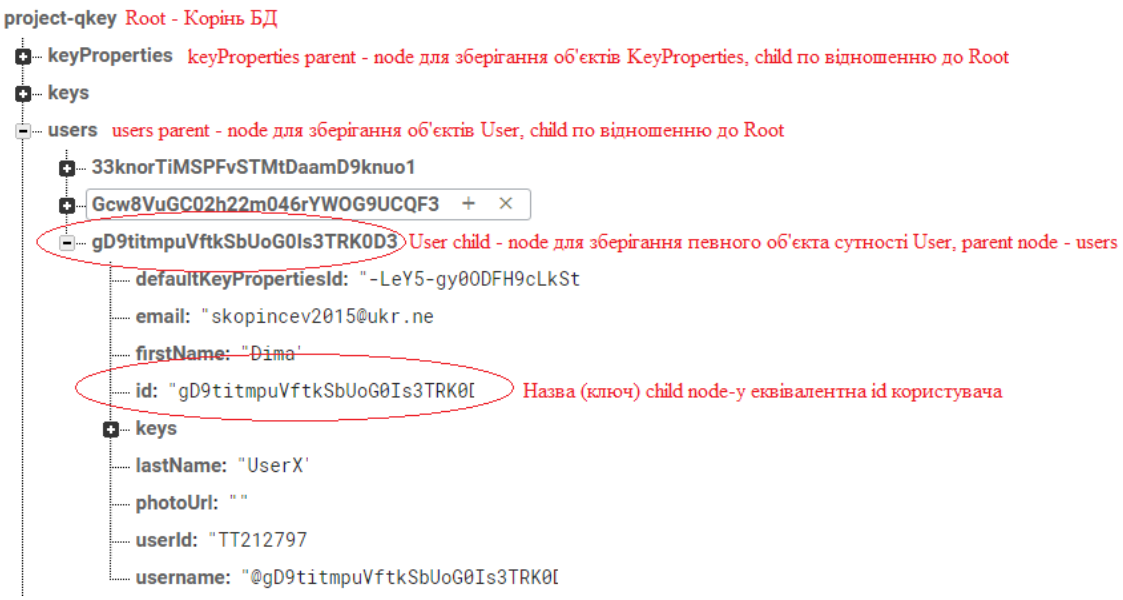


Рисунок 4.1 – Приклад реалізованої структури БД даного проекту

4.2.2 Опис засобів роботи з БД

Для роботи з RTDB було прийнято рішення створити свого роду інтерфейс-фасад, який надаватиме змогу централізовано доступатися до всіх необхідних даних не розкриваючи деталі реалізації та специфіку обраної структури БД. Його назва – **DbHelper**. В свою чергу, даний інтерфейс лише слугуватиме фасадом для уніфікації доступу до бази. Його основне завдання – передати клієнтові відповідного менеджера БД, який необхідний йому для вирішення поставленої задачі.

Для виокремлення логіки доступу до об’єктів кожної сутності, було створено п’ять інтерфейсів-менеджерів бази даних, по одному для кожної сутності:

- **UserHelper** – допоміжний інтерфейс БД для роботи із об’єктами сутності User;
- **VehicleHelper** – допоміжний інтерфейс БД для роботи із об’єктами сутності Vehicle;
- **KeyHelper** – допоміжний інтерфейс БД для роботи із об’єктами сутності Key;
- **ShareGroupHelper** – допоміжний інтерфейс БД для роботи із об’єктами сутності ShareGroup;

- **KeyPropertiesHelper** – допоміжний інтерфейс БД для роботи із об’єктами сутності KeyProperties.

Кожен з них являється незалежним інструментом для роботи з БД. Власне доставкою їх до клієнта і займається DbHelper. Не дивлячись на те що було прийнято рішення про створення окремих менеджерів БД для кожної сутності, логіка роботи із БД дуже схожа для кожної з них за рахунок простої і логічної структури RTDB, тому було розроблено базовий інсрумент **BaseHelper**, який лягає в основу інших п’яти і реалізує базовий функціонал, необхідний для більшості хелперів: генерація ключа для нового child-у, створення, оновлення та видалення запису об’єкта сутності, пошук об’єктів за унікальним ідентифікатором та будь-яким текствоим полем. Для доступу до RTDB використовується спеціальний допоміжний клас Firebase Realtime Database SDK – **DatabaseReference**.

Для уніфікації та опису структури БД у коді було створено спеціальний об’єкт **DbScheme** – він зберігає відомості про назви базових child-ів та назви певних атрибутів об’єктів сутностей, щоб надати змогу клієнтові **RTDB API** оновлювати необхідні поля об’єктів.

4.3 Опис засобів роботи зі сховищем медіа даних

Сховище медіа даних Firebase Storage працює схожим чином із базою даних. В рамках цього проекту, медіа дані представлені лише зображеннями транспортних засобів та фотографіями користувачів. Структура сховища: два головних репозиторія для даних всередині кореневого, **vehicles** – для зберігання зображень транспорту та **users** – для фотографій користувачів. Для розмежування даних всередині кожного з репозиторіїв, назви файлів відповідають унікальним ідентифікаторам транспортного засобу/юзера відповідно. Це також сприятиме полегшенню навігації сховищем. Приклад структури сховища продемонстровано на Рисунку 4.3.

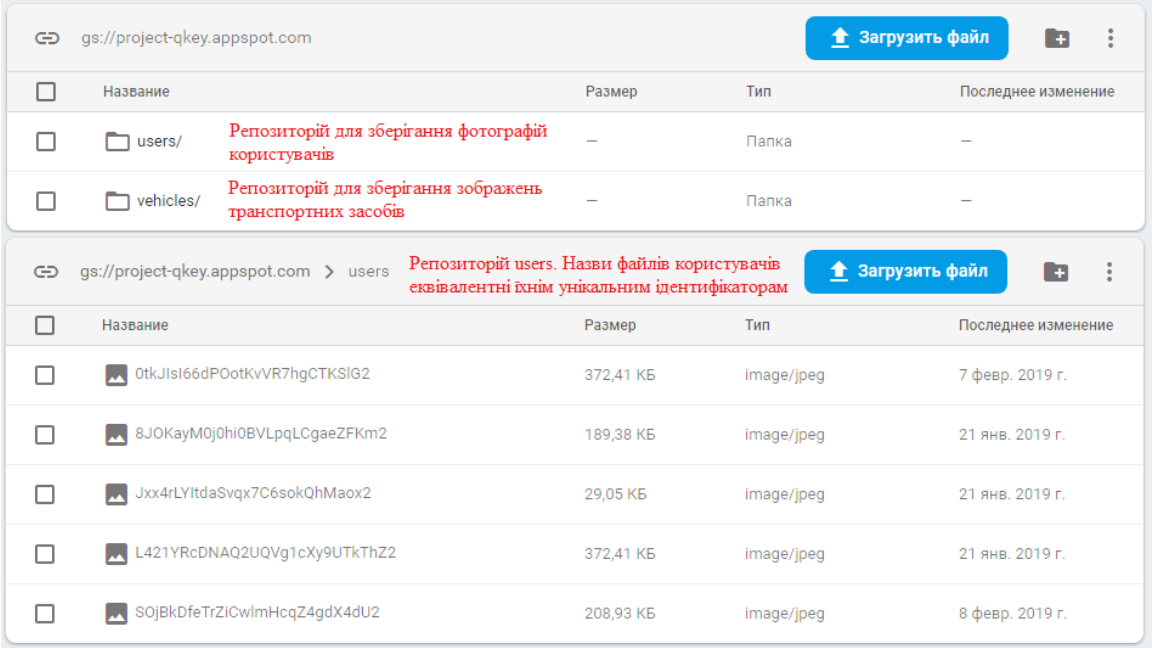


Рисунок 4.2 – Приклад структури Firebase Storage, обраної для проекту

Зважаючи на подібність структур БД та сховища, для роботи із ним було вирішено створити інструменти, подібні до інструментів, які було розроблено для роботи із RTDB, а саме: було створено інтерфейс-фасад **MediaHelper**, який займається доставкою до клієнта API необхідного йому менеджера медіа сховища. Так само було створено базовий допоміжний інтерфейс менеджера – **StorageHelper**, який, з використанням спеціального класу Firebase Storage SDK **StorageReference**, дозволяє клієнтові завантажити необхідне зображення у сховище і отримати URL посилання для його подальшого використання.

Єдина відмінність із засобами роботи з базою даних полягає у відсутності різних реалізацій менеджерів для медіа даних користувачів і транспортних засобів, адже наразі вони не передбачають імплементації особливого функціоналу.

4.4 Висновки

Під час розробки засобів та інструментів для роботи із Firebase бекендом та в цілому та сервісів і служб Firebase окремо було розроблено оптимальний, надійний і універсальний модуль для роботи із серверною частиною системи.

Частина коду даного інструментального модуля представлена у Додатку А.

5 НАЛАШТУВАННЯ СЕРВЕРНИХ КОМПОНЕНТІВ

5.1 Створення Firebase проекту

Перш за все, необхідно створити проект за допомогою веб додатку Firebase Management Console, який слугує основним інструментом налаштування серверної частини. Усі подальші кроки, зображено на Рисунках 5.1 – 5.3

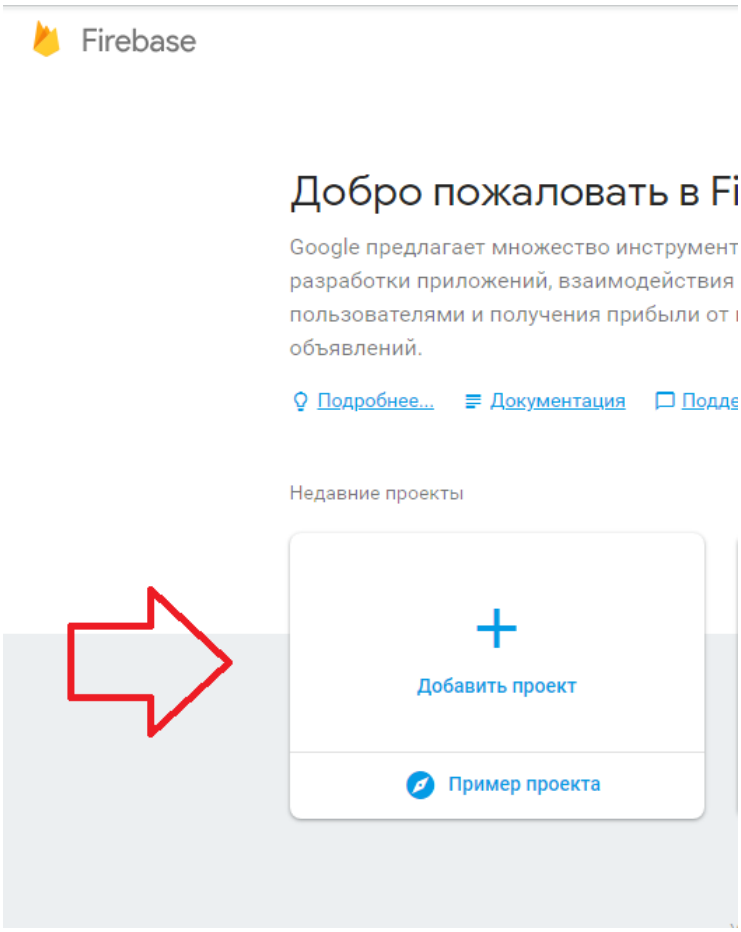


Рисунок 5.1 – Ініціюємо створення нового проекту

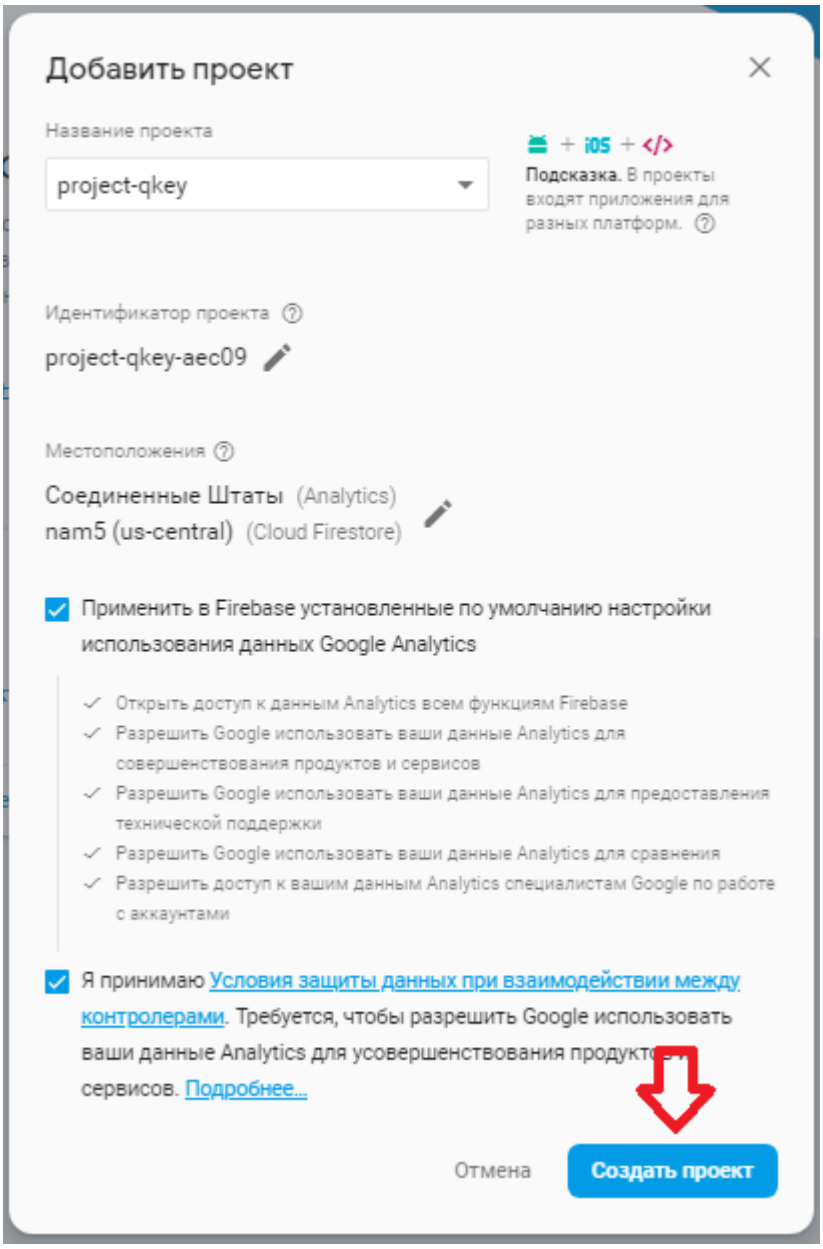


Рисунок 5.2 – Називаємо проект та створюємо його

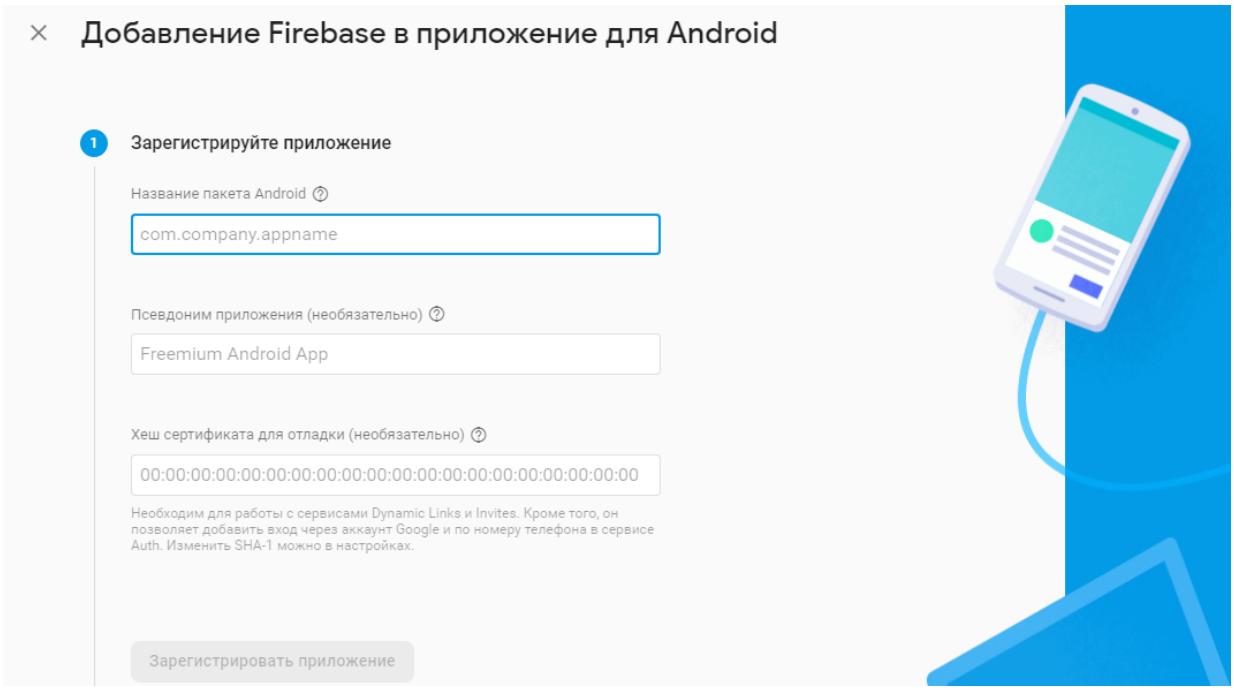


Рисунок 5.3 – Додаємо Android додаток до проекту

5.2 Налаштування сервісу аутентифікації

Наразі аутентифікація користувачів відбувається лише за електронною поштою із паролем. Тому потрібно активувати відповідний метод у Способах входу Firebase Authentication проекту. Налаштування служби зображено на Рисунку 5.4.

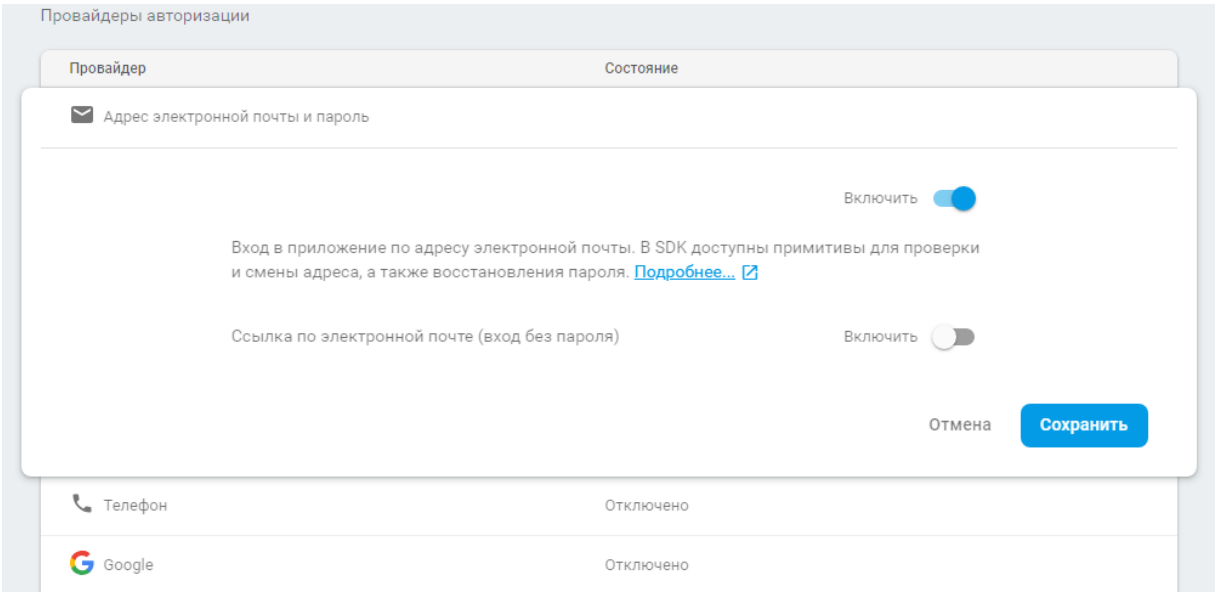


Рисунок 5.4 – Активація провайдера авторизації “Адреса електронної пошти і пароль”

5.3 Налаштування БД

Налаштування Firebase Realtime Database складається з двох етапів: власне створення RTDB та встановлення правил користування базою. Налаштування RTDB для даного проекту зображено на Рисунках 5.5 – 5.6.

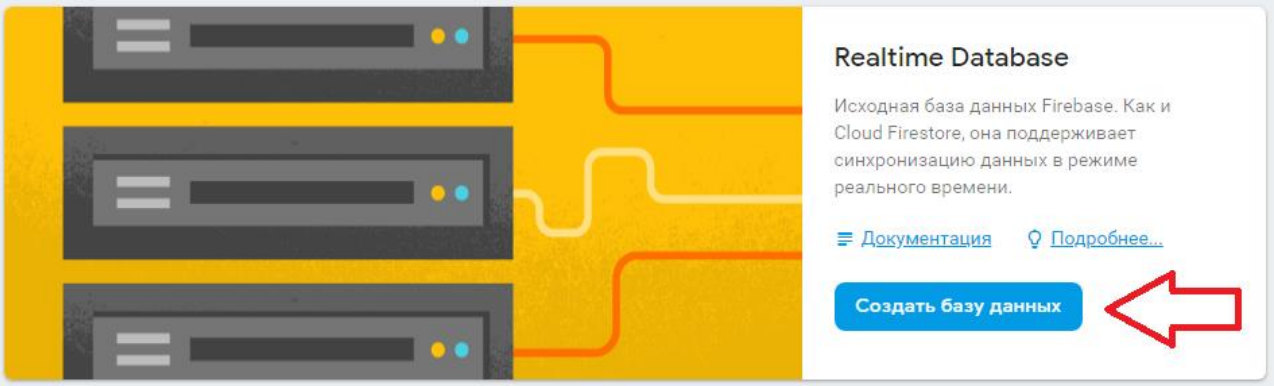


Рисунок 5.5 – Створення Realtime Database БД

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null",

    "users": {
      ".indexOn": ["username", "firstName", "lastName"]
    },

    "vehicles": {
      ".indexOn": ["vehicleId"]
    },

    "shareGroups": {
      ".indexOn": ["name", "groupId"]
    }
  }
}
```

Рисунок 5.6 – Встановлення правил користування БД

В даному випадку, правила встановлюють обмеження на зчитування та запис даних – лише для авторизованих користувачів. А також описують атрибути, за

якими індексуються об’єкти в нашій базі, що пришвидшує пошук даних за відповідними полями в майбутньому.

5.4 Налаштування сховища медіа даних

Налаштування Firebase Storage дещо схоже на налаштування RTDB. Встановлення сховища продемонстровано на Рисунках 5.7 – 5.8.

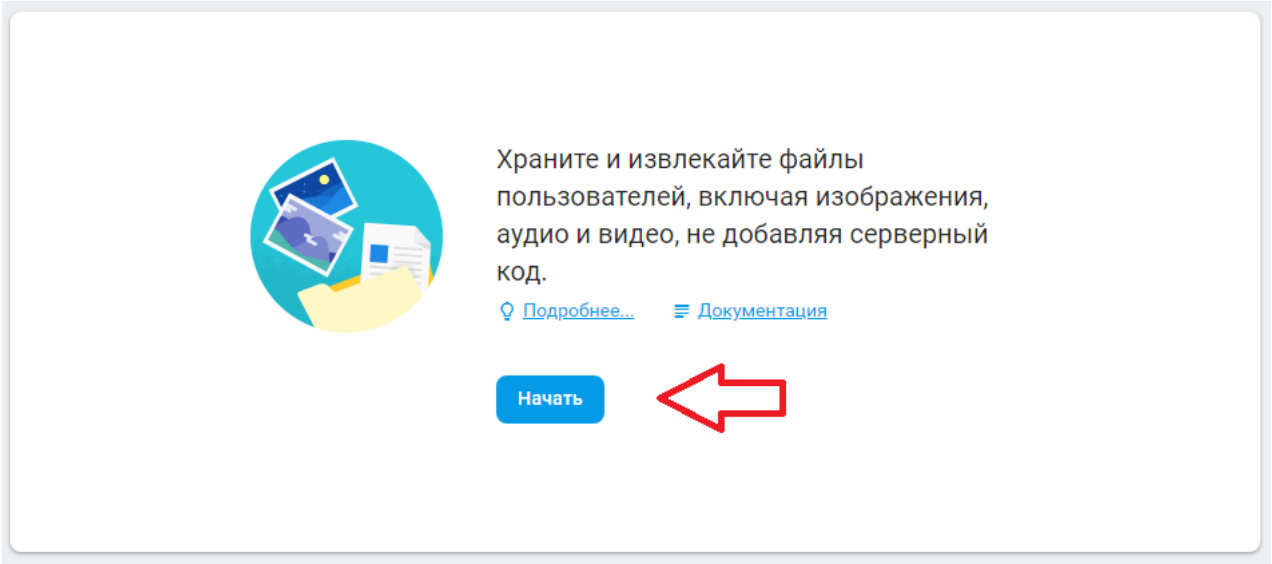


Рисунок 5.7 – Ініціалізація Firebase Storage

```
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Рисунок 5.8 – Встановлення правил користування сховищем медіа даних

В даному випадку, правила встановлють обмеження на зчитування та запис медіа даних – лише для авторизованих користувачів.

5.5 Розробка та розгортання додаткових серверних функцій

Firebase бекенд надає користувачеві багато різноманітних додаткових можливостей для модернізації власного продукту. Одним з таких засобів є **Cloud Functions** – інструмент розширення базового спектру функцій Firebase серверу за рахунок Node.js функцій, код яких завантажується на сервер і виконується лише у випадках спрацювання спеціальних тригерів, заданих розробником функції.

5.5.1 Опис Firebase Cloud Functions

Додаткові функції, розроблені для цього проекту покликані допомогти розробникові з автоматизацією обробки даних Realtime Database, проте існують й інші функції. Наприклад, **sendVerificationCode** – це Cluod Function, яка надсилає власнику транспортного засобу верифікаційний код для забезпечення двохфакторної аутентифікації під час шерінгу цифрового ключа. Інший ряд функцій, під назвою **Key properties functions** керує автоматичним створенням та видаленням дефолтних KeyProperties об’єктів для User, Vehicle, ShareGroup та Key сутностей. Надалі, задля того аби не перенасичувати даний опис, подано типовий алгоритм створення та розгортання Firebase Clound Functions і продемонстровано на Рисунках 5.9 – 5.12.

5.5.2 Розробка і розгортання Firebase Cloud Functions

Перш за все, необхідно встановити наступні компоненти:

- Node.js;
- Firebase CLI;
- Firebase tools.

Для цього завантажуємо та встановлюємо Node.js. Потім запускаємо командний рядок Node.js і встановлюємо Firebase CLI та інші Firebase інструменти командою *npm install -g firebase-tools*.

Саме після цього розпочинається власне налаштування Firebase Cloud Functions. Спершу необхідно виконати команду *firebase login*, аби увійти до свого Google/Firebase акаунту (1). Після вводу команди, командний рядок запускає стандартний браузер ПК, примушуючи юзера авторизуватися (обрати відповідний Google акаунт). Після цього, Firebase CLI просить користувача надати програмі необхідні їй для подальшої роботи дозволи. На цьому авторизація користувача завершується. За успішної авторизації, відображається відповідне повідомлення (2). Повертаючись до командного рядка, програма сповіщає про успіх (3).

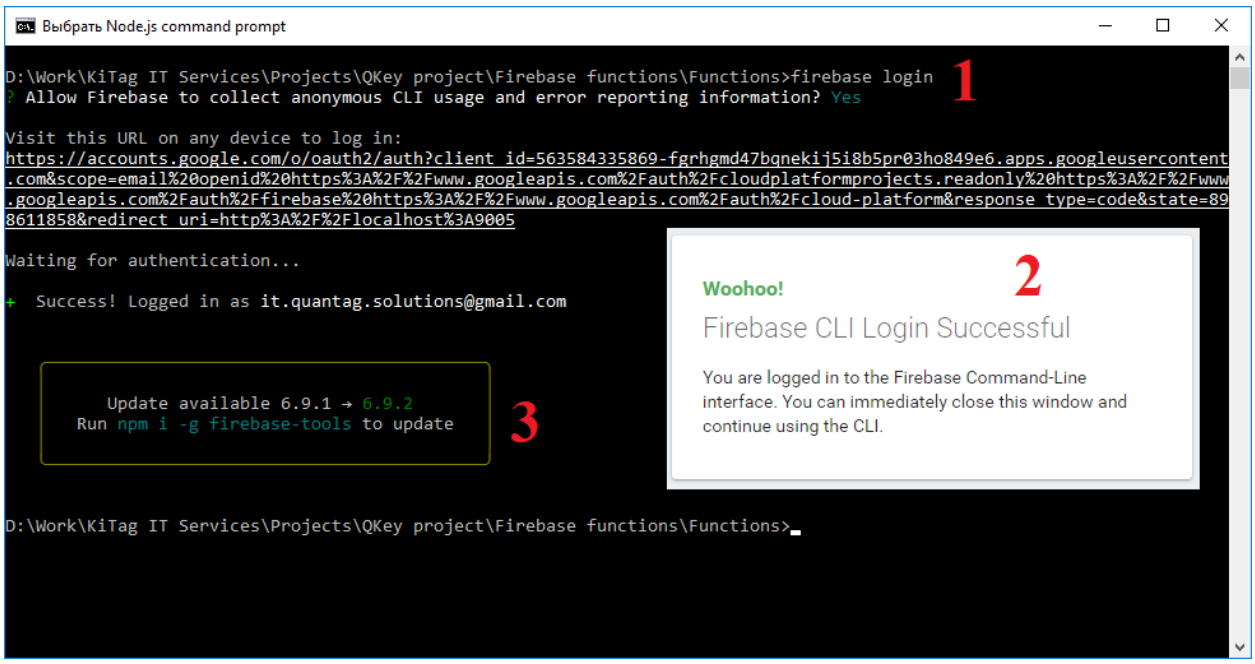


Рисунок 5.9 – Авторизація користувача для Firebase tools

Далі необхідно створити підпроект для Cloud Functions командою *firebase init functions*. Після цього Firebase CLI потребуватиме від користувача обрати встановити деякі параметри проекту, зокрема обрати мову, якою будут написані функції, JavaScript або TypeScript. Усі Cloud Functions цього проекту написані мовою програмування JavaScript. Код функцій наведений у додатку Б.

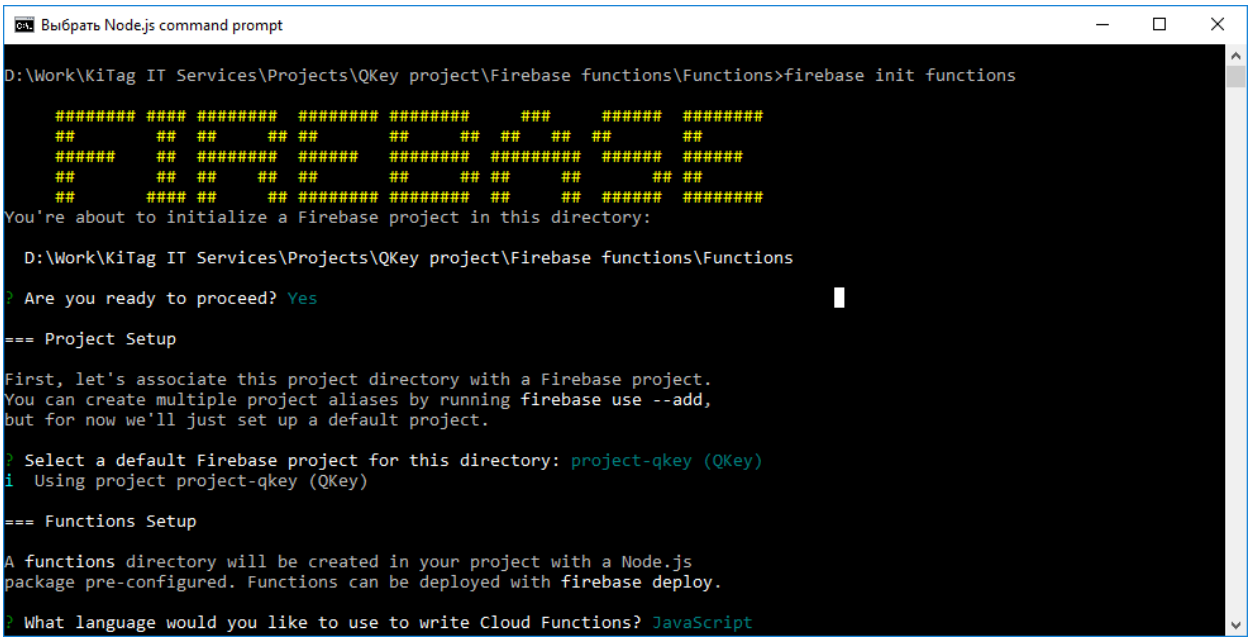


Рисунок 5.10 – Ініціалізація локального проекту для Firebase Cloud Functions

Під час ініціалізації, інструменти Firebase створюють файл **index.js** – основний файл проекту, який міститиме код функцій. Отже, програмуємо наші функції та розпочинаємо процес їх розгортання на сервері командою *firebase deploy --only functions*.

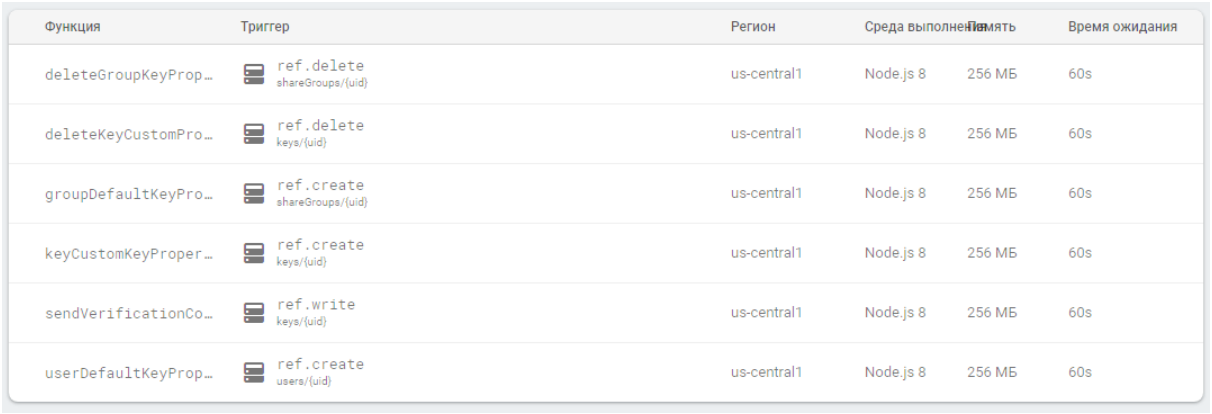


Рисунок 5.11 – Результат деплойменту Cloud Functions

5.6 Висновки

В результаті, ми отримали готовий до роботи, повністю налаштований бекенд, який надає системі у користування сервіс аутентифікації, NoSQL базу даних, сховище медіа файлів, а також широкі можливості для подальшого розширення функціоналу за допомогою Firebase Node.js Cloud Functions.

6 РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ

В термінах Android SDK, **Activity**, тобто **активність** – це компонент додатку, який займається прорисовуванням користувацького інтерфейсу. Між тим, кожну активність можна поділити на **фрагменти** – сегменти UI, які опрацьовують певний шматок логічно пов’язаного користувацького інтерфейсу.

В рамках даного проекту, для реалізації навігації у мобільному додатку між екранами використовуватиметься сучасний компонент Android SDK – **Navigation Architecture Component**. Згідно з найпопулярнішою методикою імплементації навігації ми використовуватимемо наступний підхід: кожен екран – це окремий фрагмент, а активність, яка буде хостом для даного фрагмента, міститиме усі екрани, які пов’язані логічно із даним. Тобто Activity в даному випадку – це логічно незалежний компонент додатку, що надає користувачеві ряд логічно пов’язаних функцій.

Назва мобільного додатку – **QKey**.

6.1 Розробка активності аутентифікації

Перша активність – **AuthActivity**. Є базовою активністю додатку – першою запускється під час холодного запуску додатку. AuthActivity містить екрани реєстрації, логіну, а також екран підтвердження електронної пошти. Тобто надає користувачеві увесь, необхідний для аутентифікації функціонал. Стартовий екран активності – екран логіну. Якщо користувач додатку вже залогінений, то активність зупиняється, знищується і стартує інша активність – головна, основна активність додатку.

6.1.1 Розробка екрану реєстрації

Назва екрану – **SignUpFragment**. Цей екран призначений для перевірки введених користувачем даних та реєстрації користувача у системі. Користувацький

інтерфейс екрану реєстрації можна побачити на Рисунку 6.1. Для роботи екрану, йому необхідні наступні компоненти:

- AuthManager – для реєстрації користувача та отримання даних про його запис у системі.

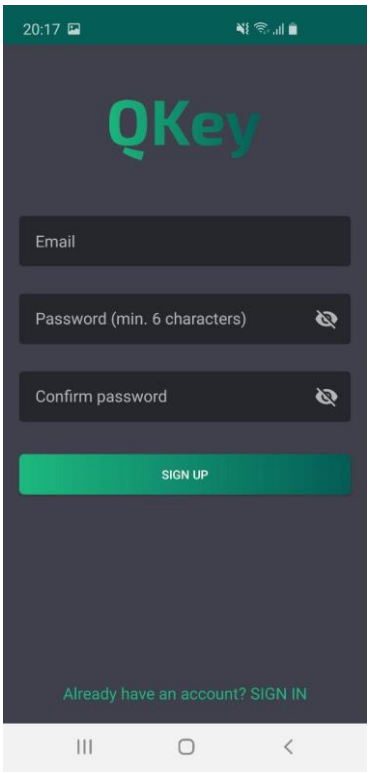


Рисунок 6.1 – Користувацький інтерфейс екрану реєстрації

В разі успішної реєстрації користувача, він буду перенаправлений до екрану підтвердження електронної пошти, в разі помилки – побачить відповідне повідомлення. Попередній у стеку екран – екран логіну.

6.1.2 Розробка екрану підтвердження електронної пошти

Назва екрану – **VerifyEmailFragment**. Цей екран призначений для надсилання користувачеві відповідного повідомлення на електронну адресу, вказану під час реєстрації, для перевірки статусу підтвердження, а також для створення запису нового користувача у БД. Користувацький інтерфейс екрану

підтвердження електронної пошти можна побачити на Рисунку 6.2. Для роботи екрану, йому необхідні наступні компоненти:

- AuthManager – для надсилання поштових повідомлень, перевірки статусу підтвердження, а також отримання даних про запис користувача у системі;
- UserHelper – для створення запису нового користувача у RTDB.

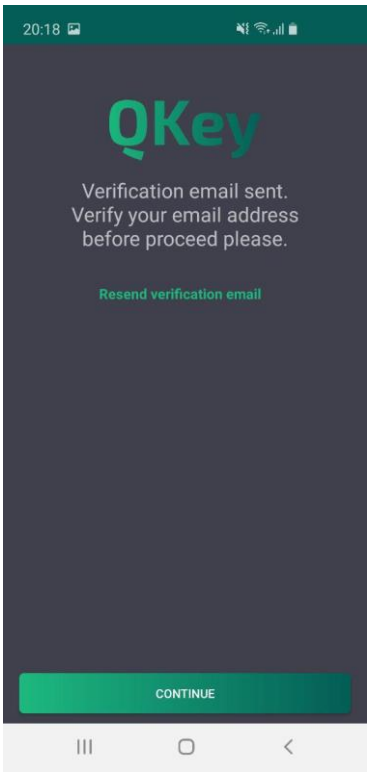


Рисунок 6.2 – Користувацький інтерфейс екрану підтвердження електронної пошти

В разі успішного підтвердження електронної адреси, дана активність зупиняється і запускається основна активність. В разі, якщо лист на електронну адресу не надійшов, користувач має змогу надіслати лист вдруге. Попередній у стеку екран – екран реєстрації.

6.1.3 Розробка екрану логіну

Назва екрану – **SignInFragment**. Цей екран призначений для перевірки введених користувачем даних, його логіну, перевірки статусу підтвердження

електронної адреси та надсилання листа зміни паролю. Користувацький інтерфейс екрану логіну можна побачити на Рисунку 6.3. Для роботи екрану, йому необхідні наступні компоненти:

- AuthManager – для надсилання поштових повідомлень, перевірки статусу підтвердження поштової адреси і логіну користувача.

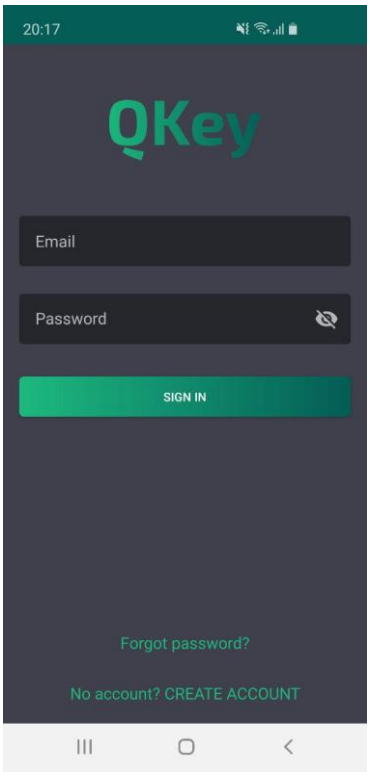


Рисунок 6.3 – Користувацький інтерфейс екрану логіну

В разі успішного підтвердження електронної адреси, дана активність зупиняється і запускається основна активність. В разі успішного відправлення листа зміни паролю, користувач отримує відповідне повідомлення. Наступний у стеку екран – екран реєстрації.

6.2 Розробка активності верифікації власника транпортного засобу

Назва активності – **OwnerVerificationActivity**. OwnerVerificationActivity містить наразі лише один екран – екран додавання транспортного засобу. Стартовий екран активності, очевидно – екран додавання транспорту.

6.2.1 Розробка екрану додавання транспортного засобу

Назва екрану – **AddVehicleFragment**. Цей екран призначений для завантаження даних про користувача, завантаження даних про транспортниій засіб, верифікацію власника та асоціацію транспорту із власником. Користувацький інтерфейс екрану додавання транспортного засобу можна побачити на Рисунку 6.4. Для роботи екрану, йому необхідні наступні компоненти:

- AuthManager – для отримання даних про запис користувача у системі;
- DbHelper – інтерфейс доступу до БД в даному випадку надає цілий ряд допоміжних менеджерів БД: UserHelper, VehicleHelper та ін. для завантаження даних та їх оновлення.

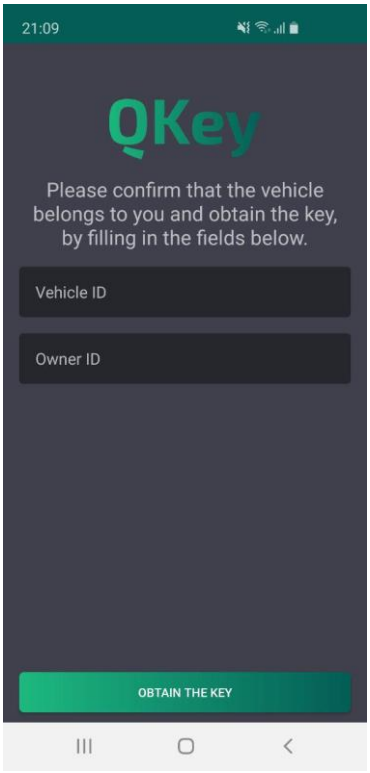


Рисунок 6.4 – Користувацький інтерфейс екрану додавання транспортного засобу

Активність разом з екраном стартує з основної активності. В разі успішної верифікації користувача як власника даного транспортного засобу, дана активність зупиняється та знову стартує основна. В результаті помилки або неуспішної верифікації – користувач отримає відповідне повідомлення. Якщо власник успішно

пройшов перевірку, то для нього створюється цифровий ключ, тим само асоціюючи його та транспортний засіб у базі даних.

6.3 Розробка основної активності

Назва активності – **MainActivity**. MainActivity містить найбільшу кількість екранів, оскільки головний фукціонал додатку міститься саме тут. Стартовий екран активності – екран транспортних засобів.

6.3.1 Розробка навігаційної панелі

Головна активність містить навігаційну панель, яка є загальною для всіх екранів MainActivity. Назва даної панелі – **NavigationDrawer**. За допомогою цього віджета можна перейти на екран транспортних засобів, екран додавання транспорту, екран друзів, екран цифрових ключів, розшарених юзеру іншими користувачами та екрану налаштувань. Також можна вийти з власного акаунту. Користувацький інтерфейс навігаційної панелі можна побачити на Рисунку 6.5.

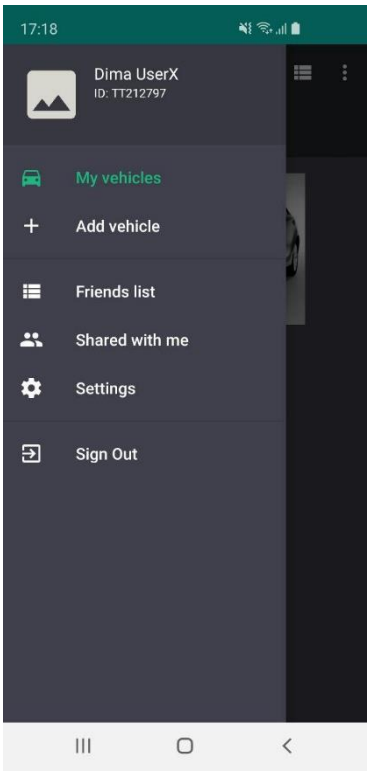


Рисунок 6.5 – Користувацький інтерфейс навігаційної панелі

6.3.2 Розробка екрану транспортних засобів

Назва екрану – **VehiclesFragment**. Цей екран призначений для відображення транспортних засобів користувача, статистики їх використання, а також їх видалення. За допомогою екранного меню можна подивитись детальну інформацію про засіб та перейти на екран шерінгу цифрового ключа. Користувацький інтерфейс екрану додавання транспортного засобу можна побачити на Рисунку 6.6. Для роботи екрану, йому необхідні наступні компоненти:

- AuthManager – для отримання даних про запис користувача у системі;
- DbHelper – інтерфейс доступу до БД в даному випадку надає цілий ряд допоміжних менеджерів БД: UserHelper, VehicleHelper, KeyHelper та ін. для завантаження даних та їх оновлення.

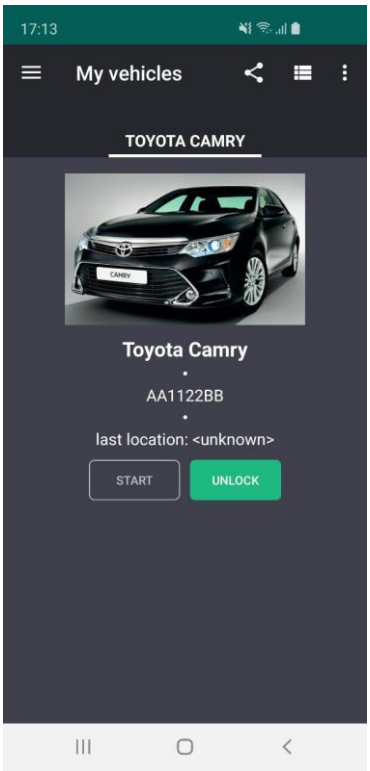


Рисунок 6.6 – Користувацький інтерфейс екрану транспортних засобів

Якщо користувач володіє декількома транспортними засобами, то він матиме можливість продивитися усі транспортні засоби на даному екрані. Ініціювати процес додавання транспорту можна запустивши активність верифікації юзера за

допомогою NavigationDrawer. В разі успішного видалення засобу, цифровий ключ, який асоціює даного користувача із даним транспортом видаляється, а засіб зникає з екрану.

6.3.3 Розробка екрану деталей транспортного засобу

Назва екрану – **VehicleDetailsFragment**. Цей екран призначений для відображення деталей про транспортний засіб, а також редагування назви і зображення транспорту. Користувацький інтерфейс екрану деталей транспортного засобу можна побачити на Рисунку 6.7. Для роботи екрану, йому необхідні наступні компоненти:

- VehicleHelper – для завантаження даних про транспортний засіб та оновлення його назви;
- MediaHelper – для завантаження та оновлення зображення засобу.

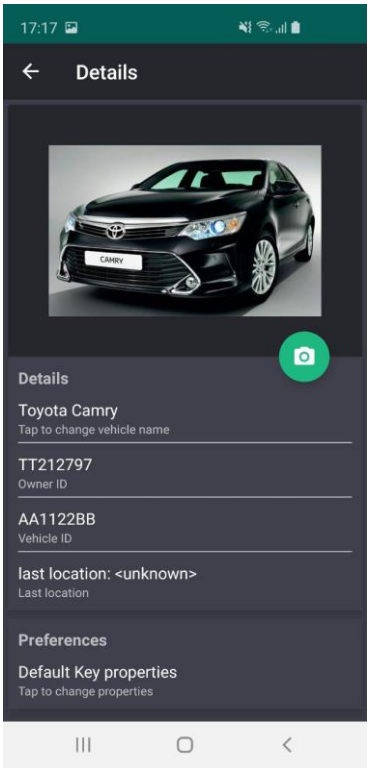


Рисунок 6.7 – Користувацький інтерфейс екрану деталей транспортного засобу

Наразі користувач має можливість змінити зображення обравши будь-яку картинку, збережену на його смартфоні. В разі успішної зміни зображення, воно одразу завантажується на сервер і після цього змінюється на екрані. Аналогічно змінюється назва транспортного засобу. Інші дані незмінні. Натискаючи на “Default Key properties”, користувач потрапляє на екран налаштування параметрів цифрових ключів даного засобу за замовчуванням. Тобто, якщо користувач обирає відповідну політику для певного розшареного цифрового ключа, він матиме властивості, які встановлені для цього транспорту по дефолту. Попередній екран у стеку – екран транспортних засобів.

6.3.4 Розробка екрану шерінгу цифрового ключа

Назва екрану – **ShareFragment**. Цей екран призначений для пошуку користувачів за ім’ям або username-ом, шерінгу цифрового ключа певному обраному користувачеві, а також для переходу на екран надсилання запрошення юзерам, якщо вони наразі не зареєстровані у системі. Користувацький інтерфейс екрану шерінгу цифрового ключа можна побачити на Рисунку 6.8. Для роботи екрану, йому необхідні наступні компоненти:

- DbHelper – для створення цифрових ключів та пошуку юзерів;
- AuthManager – для отримання даних про запис користувача у системі;
- MediaHelper – для завантаження фотографій користувачів.

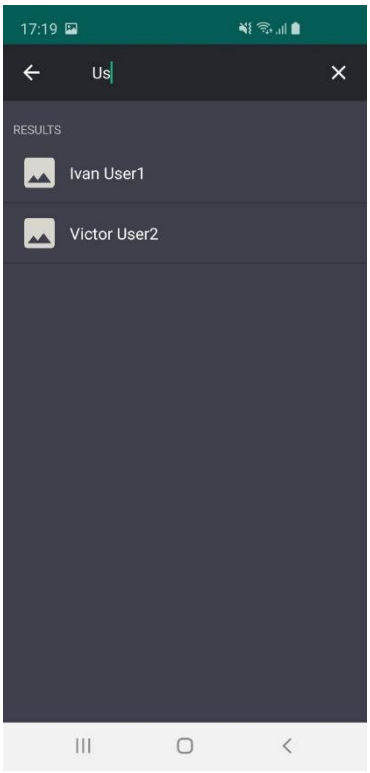


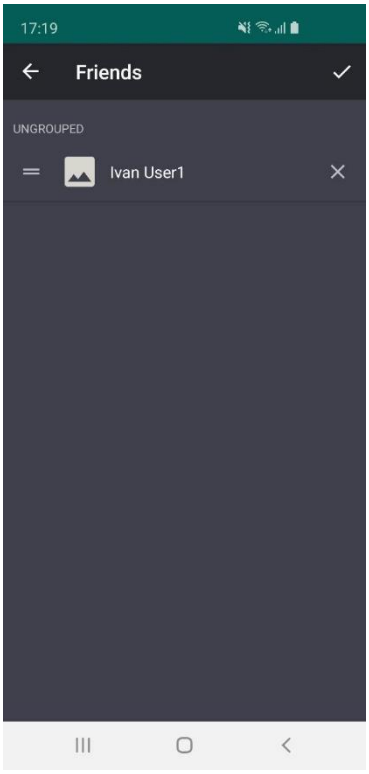
Рисунок 6.8 – Користувацький інтерфейс екрану шерінгу цифрового ключа

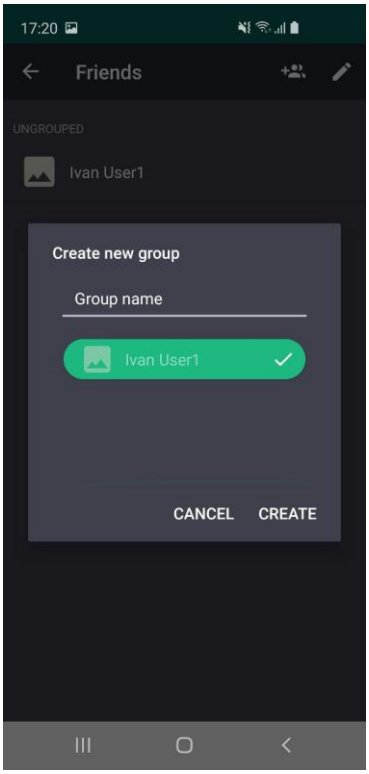
В разі успішного шерінгу, юзер отримує відповідне повідомлення, а користувач, якому надано доступ до засобу – стає другом і відображається на екрані друзів. В свою чергу, щойно створений цифровий ключ друг може побачити на екрані розшарених йому ключів. Попередній екран у стеку – екран транспортних засобів. Якщо користувач вже надавав іншому користувачеві доступ до власних транспортних засобів, йому буде одразу запропоновано розшарити ключ одному з них, тобто друзів.

6.3.5 Розробка екрану друзів

Назва екрану – **FriendsFragment**. Цей екран призначений для відображення усіх друзів користувача, видаленням друзів, створенням та управлінням груп, а також переходом на екран розшарених друзям цифрових ключів. Користувацький інтерфейс екрану друзів можна побачити на Рисунках 6.9 – 6.10. Для роботи екрану, йому необхідні наступні компоненти:

- DbHelper – для видалення друзів та відповідних ключів, створенням та редагуванням груп;
- AuthManager – для отримання даних про запис користувача у системі;
- MediaHelper – для завантаження фотографій користувачів.





Рисунки 6.9 та 6.10 відповідно – Користувацький інтерфейс екрану друзів

Якщо користувач немає жодного друга, то на екрані відображається відповідне повідомлення. В іншому випадку на цьому екрані відображено список усіх друзів користувачів, об’єднаних у блоки за групами. Друзі, які не потрапили до жодної групи помічені міткою **Ungrouped** і лише зі списку цих користувачів можна створити нову групу. За допомогою екранного меню, можна увімкнути режим редагування або запустити діалог створення нової групи, зображений на Рисунку 6.10. В разі успішного створення групи, у списку з’являється новий відповідний блок. Режим редагування зображено на Рисунку 6.9. В даному режимі користувач має змогу перемістити друга з однієї групи у іншу або видалити його взагалі. У випадку видалення, усі цифрові ключі, які були надані у користування даному другові видаляються. Наступний екран у стеку – екран дефолтних

властивостей цифрових ключів для групи та екран ключів певного друга. Екран властивостей групи аналогічний екранові дефолтних властивостей ключів транспортного засобу.

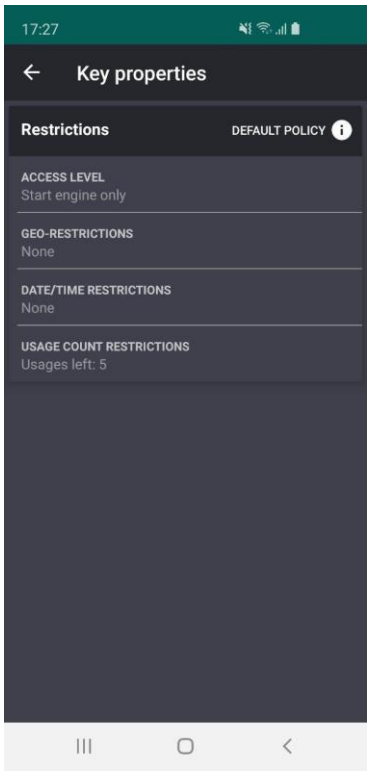
6.3.6 Розробка екрану дефолтних властивостей цифрового ключа

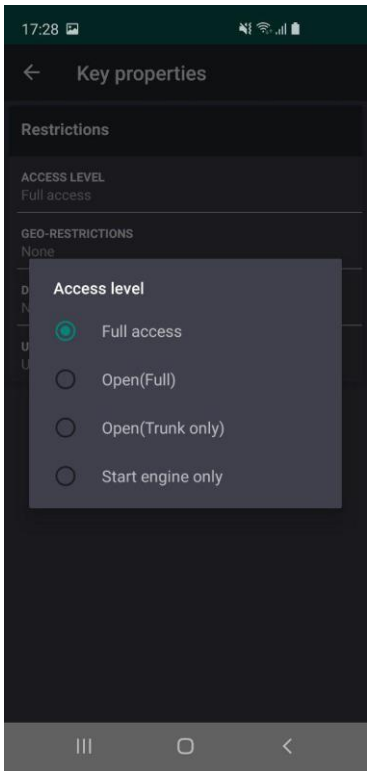
Назва екрану – **KeyPropertiesFragment**. Цей екран призначений для відображення та редагування властивостей цифрових ключів за замовченням. Насправді цей фрагмент не використовується напряму, проте являється базовим для чотирьох інших фрагментів:

- **DefaultKeyPropertiesFragment** – екран дефолтних властивостей ключів для даного користувача, доступний на екрані налаштувань;
- **VehicleKeyPropertiesFragment** – екран дефолтних властивостей ключів для даного транспортного засобу, доступний на екрані деталей транспорту;
- **GroupKeyPropertiesFragment** – екран дефолтних властивостей ключів для даної групи користувачів, доступний на екрані друзів;
- **SharedKeyPropertiesFragment** – екран властивостей конкретного цифрового ключа, доступний на екрані ключів, розшарених певному другові.

Які саме властивості буде надано ключеві, визначається його політикою. Користувацький інтерфейс екрану дефолтних властивостей цифрового ключа можна побачити на Рисунках 6.11 – 6.12. В даному випадку зображено SharedKeyPropertiesFragment. Для того, аби мати можливість редагувати його властивості, має бути обрана політика POLICY_CUSTOM. В іншому разі, редагування властивостей відбувається на відповідному екрані. Для роботи екрану, йому необхідні наступні компоненти:

- DbHelper – для завантаження властивостей та їх редагування.





Рисунки 6.11 та 6.12 відповідно – Користувацький інтерфейс екрану дефолтних властивостей цифрового ключа (SharedKeyPropertiesFragment)

Якщо політика, обрана для конкретного цифрового ключа не відповідає екранові – відображається відповідне повідомлення, а редагування його властивостей – заборонено. Наразі, редагування гео-обмежень не розробляється, оскільки прототип не оснащений модулем GPS. Також в контексті дипломного проекту не розглядається реалізація обмежень по часу/даті. В разі успішного редагування властивостей, юзер побачить відповідну зміну інтерфейсу.

6.3.7 Розробка екрану цифрових ключів

Назва екрану – **KeysFragment**. Цей екран призначений для відображення цифрових ключів. Насправді цей фрагмент не використовується напряму, проте являється базовим для трьох інших фрагментів:

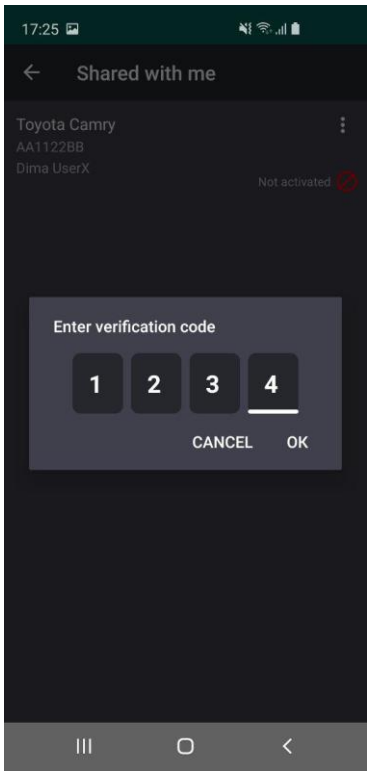
- **SharedKeysFragment** – екран цифрових ключів, які надано певному другові, доступний на екрані друзів;

- **FriendsKeysFragment** – екран цифрових ключів, які надано у користування юзеру іншими користувачами, доступний через навігаційну панель;
- **VehicleKeysFragment** – екран дефолтних властивостей ключів для даної групи користувачів, доступний на екрані друзів.

Користувацький інтерфейс екрану цифрових ключів можна побачити на Рисунках 6.13 – 6.14. В даному випадку зображено FriendsKeysFragment. Для того, аби мати можливість користуватися ключем в ролі друга, його необхідно активувати. Для роботи екрану, йому необхідні наступні компоненти:

- DbHelper – для завантаження списку ключів, для генерації нового верифікаційного коду та видалення ключа;
- AuthManager – для отриманная даних про запис користувача у системі.

Якщо наразі не існує жодного ключа, на екрані відображається відповідне повідомлення. В якості власника, користувач має змогу згенерувати новий верифікаційний код, видалити ключ та змінити його політику. В разі успішної генерації коду, юзер отримує електронного листа із кодом підтвердження та відповідне повідомлення. В якості друга, користувач має змогу активувати цифровий ключ, запустивши діалог активації, зображений на Рисунку 6.14, та видалити ключ, відмовившись від нього. В разі успішної активації, юзер побачить відповідні зміни користувацького інтерфейсу. Усі дії над ключем, доступні користувачеві, незалежно від його ролі запускаються через контекстне меню відповідного ключа.



Рисунки 6.13 та 6.14 відповідно – Користувацький інтерфейс екрану цифрових ключів (FriendsKeysFragment)

6.3.8 Розробка екрану налаштувань

Назва екрану – **SettingsFragment**. Цей екран призначений для редагування інформації про користувача, його фотографії, а також переходу на екран властивостей цифрових ключів за замовчуванням для даного юзера. Користувацький інтерфейс екрану налаштувань можна побачити на Рисунку 6.15. Для роботи екрану, йому необхідні наступні компоненти:

- DbHelper – для завантаження та оновлення інформації про юзера;
- AuthManager – для отримання даних про запис користувача у системі.
- MediaHelper – для завантаження та оновлення фотографії користувача.

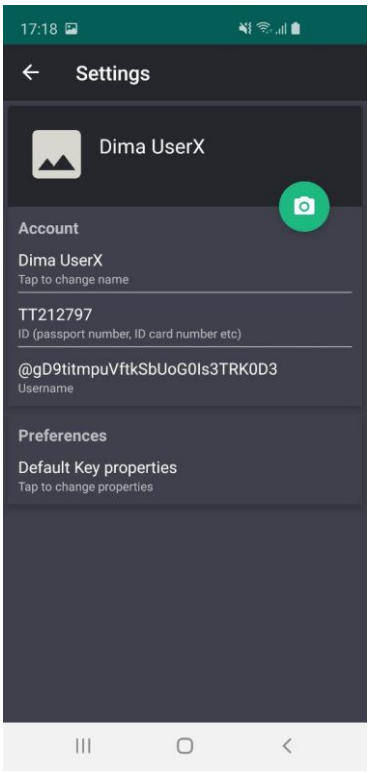


Рисунок 6.15 – Користувацький інтерфейс екрану налаштувань

Користувач має змогу змінити власне фото, а також прізвище, ім’я і username. В разі успішного оновлення інформації або фото, на екрані відображаються відповідні зміни користувацького інтерфейсу та повідомлення про успіх. Для успішної зміни username-у він має бути унікальним серед усіх користувачів, зареєстрованих у системі. Перехід на цей екран відбувається через навігаційну панель.

6.4 Розробка BLE сервісу

Для того, щоб забезпечити комунікацію мобільного додатку із IoT девайсом, який представляє транспортний засіб, було вирішено створити службу під назвою UserBleService. Служба, або сервіс Android SDK – це компонент додатку, який може виконувати операції у фоновому режимі. UserBleService ініціюється під час запуску MainActivity та знищується разом із нею. Головна задача UserBleService – це створення та запуск BLE GATT сервера із QKey сервісом, який надаватиме системі можливість обмінюватися даними між смартфоном та IoT девайсом.

Для того, щоб створити QKey сервіс, було розроблено допоміжний клас QKeyBleProfile, який створює сервіс із двома GATT характеристиками:

- DATA – для зчитування даних з QKey GATT серверу;
- MESSAGE – для запису даних до QKey GATT серверу.

6.5 Висновки

В результаті роботи над мобільним додатком, було розроблено складну багатокомпоненту програму для операційної системи Android, яка надає користувачеві широкий функціонал для менеджменту цифрових ключів для його транспортного засобу.

Частина коду модуля даного додатку представлена у Додатку В.

7 РОЗРОБКА ДОДАТКУ ТРАНСПОРТНОГО ЗАСОБУ

7.1 Налаштування модуля додатку

Як було зазначено раніше, додаток створюється для операційної системи Android Things, розробленої та підтримуваної компанією Google.

Android Things являє собою відгалудження від відомої ОС Android, а розробка додатків для неї дуже схожа на розробку додатків для її предка. Незважаючи на схожість, звісно обидві операційні системи мають ряд відмінностей. Далі наведено основні відмінності:

- **Робота з периферією.** IoT пристрої призначені для створення системи, яка складається із невизначеної кількості девайсів, які можуть дуже відрізнятися від стандартної периферії смартфонів чи планшетів під керуванням Android. Зважаючи на це, можливості системи були значно розширені, до SDK ОС Android Things було додано спеціальне API для роботи із периферією в залежності від типу її підключення. Наразі Android Things підтримує такі інтерфейси підключення: GPIO, PWM, I2C, SPI, UART;
- **Служби Play Services (Google).** На жаль, ряд підтримуваних операційною системою Android Things сервісів Google значно менший за аналогічний у оригінальної ОС. Проте цього функціоналу більш ніж достатньо, адже в нашому випадку використовуються лише служби, пов’язані із Firebase, які підтримуються системою повною мірою.

Версія ОС Android Things, яка використовувалась під час розробки – **1.0.11 (API Level 1)**.

Версія ОС Android, ядро Android Things: **8.1 (API Level: 27)**.

Додаток написаний більшим чином мовою Kotlin, проте деякий функціонал імплементований на Java.

Склад додатку: одна активність, HomeActivity, яка являється стартовою точкою і запускає додаток одразу після запуску ОС, та одна служба, яка призначена для роботи із Bluetooth.

Для роботи модуля йому необхідні наступні бібліотеки:

- **com.google.android.things:androidthings:1.0** – містить SDK ОС;
- **com.google.android.things.contrib:driver-pwmservo:1.0** – містить драйвер для керування замком, детальніше буде розглянуто у наступному розділі;
- **firebase модуль** – модуль для роботи із Firebase бекендом, його розробку описано у розділі 4;
- **core модуль** – легковажний модуль, який містить спільну для обох додатків логіку.

7.2 Розробка головної активності

Активність HomeActivity призначена для старту додатка, ініціалізації усіх його головних компонентів та відображення логів цілої програми. За наявності дисплею під'єднаного до IoT девайсу, відображається екран, який являє собою просто велике текстове поле, до якого записуються логи. Для того, аби забезпечити можливість логування з будь-якої точки додатка було створено інтерфейс **Logger** з єдиним прости методом log, який приймає на вхід tag – мітку для ідентифікації місця виклику та власне лог-повідомлення. Єдиний логгер додатку було створено за паттерном Одинак для уніфікованого доступу до логу. Поточна реалізація **TextViewLogger** – це логгер, який записує повідомлення просто у текстове поле HomeActivity.

7.3 Розробка контроллера керування замком транспортного засобу

У зв'язку з тим що наразі розробляється лише прототип майбутньої системи, потрібно було заздалегідь врахувати цей факт під час роботи над додатком транспортного засобу. Саме тому, для роботи із замком транспортного засобу, було

розроблено універсальний інтерфейс LockController, який описує наступний функціонал:

- toggleLock() – функція запиту на зміни стану замка на протилежний;
- lock() – функція запиту на розблокування замка;
- unlock() – функція запиту на блокування замка.

Реалізація цього інтерфейсу в контексті даного прототипу – ServoLockController використовує драйвер для замка для керування ним. Наразі цей контроллер просто повертає обраний нами сервопривод на відповідний кут. Кут 0 градусів було прийнято за стан “Відчинено”, а кут 90 градусів за стан “Зачинено”.

Замок ініціалізується під час запуску додатку, головною активністю, станом “Зачинено”.

7.4 Розробка компонентів керування Bluetooth-ом

Як було зазначено раніше, на даний момент девайси комунікують між собою лише за допомогою технології Bluetooth, конкретніше Bluetooth Low Energy. Для того щоб змусити девайс злагоджено комунікувати зі смартфоном за технологією BLE необхідно було розробити цілу низку компонентів, які утворюють загальну інфраструктуру.

7.4.1 Сервіс BLE

Центральним компонентом даної підсистеми є VehicleBleService – служба Android, компонент додатку, який ініціалізується головною активність під час запуску додатку. Даний компонент керує усім циклом BLE.

Робота сервісу починається з ініціалізації основних засобів доступу до Bluetooth модуля мікрокомп’ютера: **BluetoothManager** та **BluetoothAdapter**. Адаптер запускається у випадку, якщо його вимкнено, налаштовується його назва у мережі. Після цього починається сканування мережі.

7.4.2 Задача пошуку QKey девайсів

Сканування мережі відбувається за допомогою іншого важливого елемента Android ОС – AsyncTask, який призначений для виконання певної задачі. Вибір AsyncTask зумовлений тим, що задача пошуку QKey девайса у мережі Bluetooth являється асинхронною, а даний компонент виконує роботу на допоміжному потоці процесу, що дозволяє не блокувати основний потік додатку. Завдяки тому, що технологія Bluetooth Low Energy дозволяє описати сервіс на BLE сервері, тобто смартфоні у нашому випадку, під час запуску задачі ми можемо одразу встановити параметри сканування, тим самим автоматично виключивши із переліку можливих BLE девайсів ті, які не мають жодного відношення до нашої системи.

Отже пошук девайсів відбувається за наступним алгоритмом: на початку робиться пауза довжиною в два періоди сканування, яка призначена для запобігання хаотичному під'єднанню/від'єднанню девайсів, після цього починається сканування ефіру на предмет BLE GATT серверів, що містять службу QKey, із періодом в 500 міллісекунд. Коли девайс знайдено, відбується перевірка відстані між пристроями за допомогою грубої оцінки на основі потужності отриманого сигналу. Деталі алгоритму можна побачити на блок-схемі.

7.4.3 Засоби оцінки відстані між BLE пристроями

Оцінка відстані відбувається за наступною формулою:

$$\mathrm{d} = 10^{\frac{(M-S)}{(10\times \mathrm{N})}}, \text{де} \qquad (7.1)$$

- **M** (measured power) – тобто константа, яка описує потужність BLE сигналу, яка очікується на відстані одного метра між смартфоном і мікрокомп'ютером. Вимірюється в дБм (децибел щодо 1 мілівата). Наразі встановлено значення **-90 дБм**;

- **S** – реальна потужність сигналу, що була зареєстрована після виявлення девайса;
- **N** – коефіцієнт, який класифікує середовище за загальною кількістю завад. Знаходиться в межах від 2 до 4. Якщо оціночна відстань сильно відрізняється від реальної, слід збільшити цей коефіцієнт. Експериментальним шляхом було виявлено, що оптимальне значення цього коефіцієнта – 3;
- **d** – результат, оціночна відстань у метрах.

Слід зауважити, що така оцінка відстані – дуже груба і не дає змогу реально порахувати відстань між пристроями, адже дуже сильно залежить від усіх типів завад, які впливаюсь на потужність сигналу. Для того, щоб трохи компенсувати цей недолік даного методу, використовується наступний фільтр:

$$S_{\text{кор}} = \alpha \times S_i + (1 - \alpha) \times S_{i-1}\text{, де} \qquad (7.2)$$

- S_i – потужність сигналу, яка була отримана під час виявлення пристрою в мережі;
- S_{i-1} – потужність сигналу, яка була обрахована попереднього разу;
- α – коефіцієнт фільтрації. Використовується значення 0,75;
- $S_{\text{кор}}$ – кореговане значення потужності сигналу за допомогою нашого фільтра.

Наразі встановлено мінімальне значення оціночної відстані у **30 сантиметрів**. Така оцінка дає змогу використовувати Bluetooth Low Energy технологію схожим чином із NFC, що позитивно впливає на користувацький досвід та виключає випадкові з’єднання із іншими смартфонами QKey системи.

7.4.4 VehicleGattServerCallback

Після того, як QKey смартфон було виявлено на достатній відстані від IoT пристрою – починається з’єдання із GATT сервером, який розгорнутно на

смартфоні. Тут на допомогу приходить **VehicleGattServerCallback** – допоміжний компонент VehicleBleService-у, який отримує безпосередньо займається опрацюванням відповідей від QKey GATT серверу.

Задача даного компонента полягає у відслідковуванні стану з'єднання, ініціалізації пошуку необхідного QKey сервісу на GATT сервері, опрацюванні відповідей стосовно результату зчитування та запису даних у сервіс.

Зворотній зв'язок із сервісом імплементовано за допомогою ServerConnectionCallback – це простий інтерфейс, який сповіщає BLE службу про зміну стану з'єднання, про результати пошуку сервісів на сервері, а також про результати запису/зчитування даних QKey сервісу.

В разі успішного встановлення з'єднання між пристроями, починається пошук відповідного сервіса на GATT сервері. Слід сказати про те, що сервіс в будь-якому разі існує, адже його унікальний ідентифікатор був встановлений у налаштуваннях при скануванні девайсів. Коли сервіс буде знайдено, служба VehicleBleService ініціює процес зчитування даних з сервісу, а саме унікального ідентифікатора користувача, якому належить смартфон, тобто залогіненого користувача. Коли дані буде прочитано, Bluetooth Low Energy служба отримує відповідне повідомлення від VehicleGattServerCallback за допомогою ServerConnectionListener-у.

7.5 Розробка менеджера цифрових ключів

Надалі у гру вступає QKeyDigitalKeyManager – менеджер цифрових ключів QKey, який реалізує простий інтерфейс DigitalKeyManager із наступним функціоналом:

- **checkUserAccess(userId)** – функція, яка ініціює повну перевірку користувача на наявність відповідного цифрового ключа, його стан та властивості;

- **onKeyUsed()** – функція, яка сповіщає систему про те, що відповідний цифровий ключ було використано для доступу до даного транспортного засобу.

Для реалізації даного функціоналу, QKeyDigitalKeyManager-у використовує цілу низку інструментів. Перелік основних наведено нижче:

- **DbHelper** – хелпер для доступу до БД, розробка якого була описана у розділі 4. Він використовується для завантаження цифрового ключа та його властивостей;
- **VehicleUtils** – утиліта для роботи із даними, які властиві транспортним засобам. В даному випадку, вона надає менеджерові у користування унікальний ідентифікатор даного транспорту;
- **SharedPreferencesManager** – менеджер локальних прихованих даних даного додатку. В даному випадку він використовується для зберігання даних про використання цифрового ключа.

QKeyDigitalKeyManager наразі перевіряє лише наступні параметри: стан цифрового ключа та певні його властивості, а саме: рівень доступу та кількість доступних користувачеві разів використання транспоротного засобу. Перевірка цих даних виконується лише в тому випадку, якщо користувач не є власником даного транспорту, адже вважається, що власник має необмежений доступ. В майбутньому також планується перевірка часових та гео-обмежень.

7.6 Висновки

В ході роботи над даним модулем, було створено IoT додаток для Raspberry Pi 3 B під управлінням операційної системи Android Things. В результаті було отримано компонент системи QKey, який імітує реальний транспортний засіб та його механізм закривання/відкривання за рахунок сервоприводу.

Частина коду модуля даного додатку представлена у Додатку Г.

8 РОЗРОБКА ПРОТОТИПУ

8.1 Налаштування одноплатного мікрокомп’ютера

Як було зазначено раніше, в якості центрального пристрою прототипу було обрано одноплатний мікрокомп’ютер Raspberry Pi 3, модель B. Даний вибір обумовлений простими міркуваннями: Android Things підтримує лише невелику кількість апартного забезпечення, включаючи Raspberry Pi, а вибір саме цієї IoT платформи обумовлений її популярністю та кількістю напрацювань, що дало б змогу полегшити процес розробки прототипу.

Рисунок 8.1 – Одноплатний мікрокомп’ютер Raspberry Pi 3 B

Технічні характеристики даного IoT девайсу:

- Процессор (CPU) – чотирьохядерний ARM Cortex A53, тактова частота – 1.2 Гц;
- Оперативна пам’ять (RAM) – Broadcom BCM2837;
- Графічний процессор (GPU) – VideoCore IV;
- Пам’ять (Storage) – MicroSD card slot;

- Дисплей – HDMI;
- Камера – RPi Camera module v2;
- Аудіо – USB 2.0, 3.5mm Analog Output;
- Інтерфейси підключення – UART, I2C, SPI, PWM, GPIO
- Комунікація (Networking) – 10/100 Ethernet, Wi-Fi 802.11n (2.4GHz), Bluetooth® 4.1
- USB – 4x USB 2.0 Host
- Розміри – 85 мм X 56 мм

З огляду на технічні характеристики даної IoT платформи, можна одразу визначити, що вона є ідеальним кандидатом на створення прототипу.

Під час розробки були задіяні наступні активи девайсу:

- **GPIO інтерфейс** – для підключення сервоприводу до мікрокомп'ютера, який імітує в даному випадку замок транспортного засобу;
- **Bluetooth 4.1 Low Energy** – для комунікації із смартфонами;
- **Wi-Fi 802.11n** – для бездротового доступу до мережі Інтернет, який використовується для обміну даними із Firebase бекендом;
- **MicroSD card slot** – для завантаження ОС Android Things та зберігання додатку і суміжних даних. Було обрано MicroSD картку об'ємом 32 Гб;
- **USB 2.0** – для використання комп'ютерної миші. Використовувалась лише в цілях розробки, не має практичного значення;
- **HDMI роз'єм** – важливий актив, який було використано для тестування додатку, а також базового налаштування операційної системи разом із мишкою. Незважаючи на те, що практичного значення даний актив не несе, проте він є дуже важливим в процесі розробки. Фактично, тестування додатку транспортного засобу неможливо без наявності користувацького інтерфейсу, зображеного на екрані завдяки HDMI.

Щоб запустити Raspberry Pi потрібно під'єднати її до джерела напруги. Для цього було придбано блок живлення MicroUSB 5 В/3 А, обладнаний кнопкою для вмикання/вимикання живлення.

Для того, щоб розпочати роботу із Raspberry Pi, починаємо налаштування ОС. Для цього використаємо програму Android Things Setup Utility для Windows OS, розроблену компанією Google для спрощення процесу встановлення операційної системи Android Things.

Рисунок 8.2 – Блок живлення для Raspberry Pi

Отже, завантажуємо дану утиліту, вставляємо до кард-рідеру комп'ютера картку MicroSD, яка міститиме образ ОС та запускаємо програму.

```
Android Things Setup Utility (version 1.0.12)
==========================
This tool will help you install Android Things on your board and set up Wifi.

What do you want to do?
1 - Install Android Things and optionally set up Wifi
2 - Set up Wifi on an existing Android Things device
1
What hardware are you using?
1 - Raspberry Pi 3
2 - NXP Pico i.MX7D
3 - NXP Pico i.MX6UL
1
You chose Raspberry Pi 3.

Setting up required tools...
Fetching additional configuration...
Downloading platform tools...
File already downloaded.
Unzipping platform tools...
Finished setting up required tools.

Do you want to use the default image or a custom image?
1 - Default image: Used for development purposes. No access to the Android
Things Console features such as metrics, crash reports, and OTA updates.
2 - Custom image: Provide your own image, enter the path to an image generated
and from the Android Things Console.
2
Please enter the absolute path to the zip file containing your image:
C:/Users/Mateusz/Desktop/RPI3_Raspberry_Pi_3_1_userdebug_build.zip
```

Рисунок 8.3 – Початок роботи із Android Things Setup Utility

Як бачимо, на початку роботи, утиліта збирає дані про ціль користувача, його апаратне забезпечення, тип встановлюваної ОС та ін.

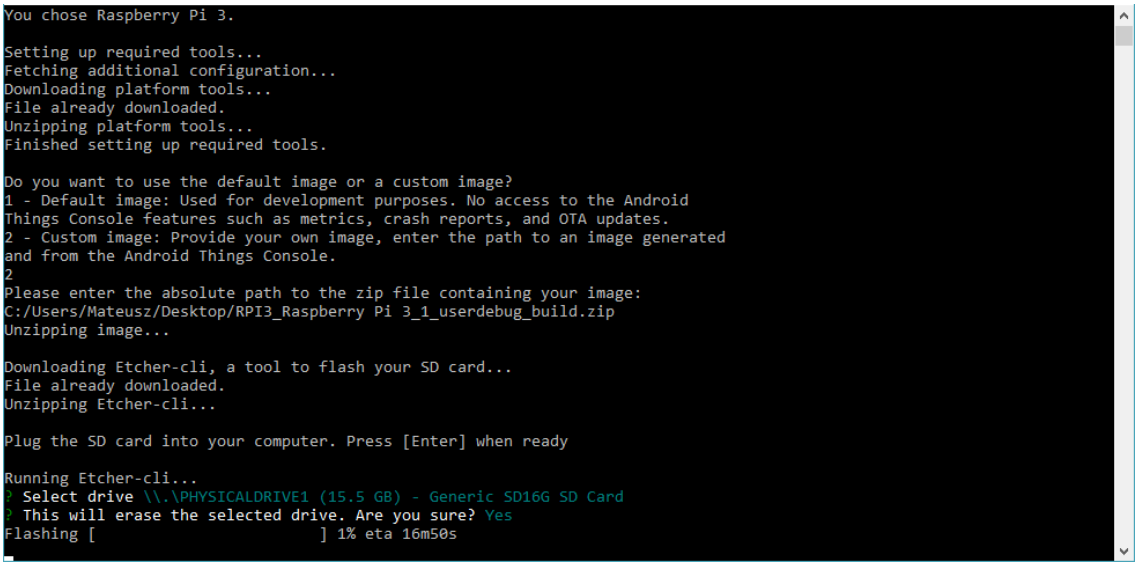


Рисунок 8.4 – Завершальний етап встановлення ОС на картку MicroSD

На останньому кроці, програма просить користувача вставити MicroSD картку, обрати який саме диск (картка) буде використана для завантаження образу ОС та ініціює процес встановлення.

Для завершення початкового налаштування ОС потрібно під’єднати девайс до мережі Інтернет, щоб можна було встановлювати додаток онлайн, а також взаємодіяти із Firebase-ом. Проте остання версія ОС, яка була використана під час роботи над даним проектом, 1.0.11, вимагає від користувача додаткоих дій. Слід

встановити мобільний додаток Android Things Console на смартфон під керуванням Android та залогінитись до консолі за допомогою акаунта Google, який використовується під час розробки.

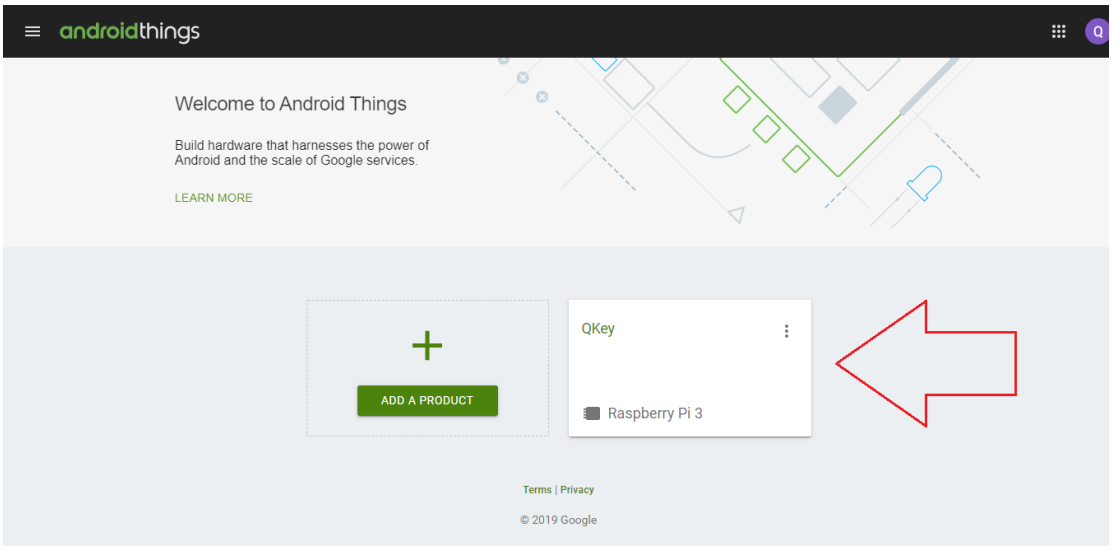


Рисунок 8.5 – Налаштування Android Things Console

В процесі налаштування Android Things Console, слід створити продукт та додати до нього наш девайс.

Потім необхідно залогінитись у аналогічному мобільному додатку, попередньо під’єднавши смартфон до тієї мережі Wi-Fi, до якої планується під’єднання Raspberry Pi.

Рисунок 8.6 – Мобільний додаток Android Things Console

Після того, як додаток віднайде в мережі наш IoT девайс користувачу буде запропоновано під’єднатись до мережі, але попередньо потрібно буде ввести згенерований ОС Android Things унікальний код для забезпечення захисту.

На цьому етапі налаштування ОС завершується.

8.2 Імітація замка транспортного засобу

В ході розробки прототипу, в якості імітатора замка реального транспортного засобу було обрано простий сервопривод, адже наразі потрібно лише демонструвати можливості системи, а інтеграція із реальним транспортним засобом буде здійснена під час створення справжнього, а не демонстраційного пристрою.

Такий вибір обумовлений простотою його інтеграції, а також низькою ціною.

8.2.1 Сервопривод Tower Pro SG90

Конкретно було обрано сервопривод Tower Pro SG90. Його вибір обумовлений гарними відгуками, дуже низькою ціною, простотою підключення, а також широким розповсюдженням в Україні.

Рисунок 8.7 – Сервопривод Tower Pro SG90

Технічні характеристики даного сервоприводу:

- Робоча напруга – 4.8 ~ 6.0 В;
- Робоча швидкість – 0.12 с/60 градусів (4.8 В) ~ 0.1 с/60 градусів (6.0 В);
- Обертальний момент – 1.6 кг/см (4.8 В);
- Кутовий діапазон – 0 ~ 180 градусів;
- Тип сервоприводу – Аналоговий, ШІМ;
- Робоча температура – -30 ~ +60 ℃;
- Ширина ШІМ імпульсу – 500 ~ 2400 мс;
- Розміри – 23.0 мм X 12.2 мм X 29.0 мм.

8.2.2 Розробка контроллера для сервоприводу

Можливості контроллера ServoLockController було описано у попередньому розділі. Також було сказано про те, що для керування сервоприводом, контроллер використовує певний драйвер. Розглянемо його докладніше.

ОС Android Things використовує концепцію user-space driver, яка полягає у тому, що драйвер розширює стандартні служби Android для керування периферією. Перевага такого підходу очевидна – програмістові не потрібно перейматись низькорівневою обробкою апартного забезпечення і дає змогу зусередитись на продуктових задачах проекту.

Ініціалізація драйверу починається з налаштування об’єкту Servo Android Things SDK, а саме встановлення назви-номеру GPIO піну, через який передається ШІМ-сигнал, діпазону ширини ШІМ сигналу, а також діапазону можливих значень кута обертання. Для передачі ШІМ сигналу було обрано GPIO пін “PWM0”, інші параметри відповідають технічним характеристикам. Діаграма GPIO пінів зображена на Рисунку 8.8. Детальна схема підключення зображена на електричній структурній схемі.

Драйвер надає контроллеру у користування наступний функціонал:

- rotate(angle) – функція повороту приводу на відповідний кут (градуси);

- getCurrentAngle() – повертає поточне значення кута сервоприводу;
- clear() – функція очищення зайнятих ресурсів;

У попередньому розділі було сказано, що при розробці контроллера ServoLockController було прийнято значення кута сервоприводу 0 градусів за стан “Відчинено”, а кут 90 градусів за стан “Зачинено”. Але слід зауважити, що під час тестування приводу, було з’ясовано, що він погано відкалібрований і реальному значенню кута 90 градусів, відповідає програмне значення 70. Таким чином компенсується неточність даного екземпляру Tower Pro SG90.

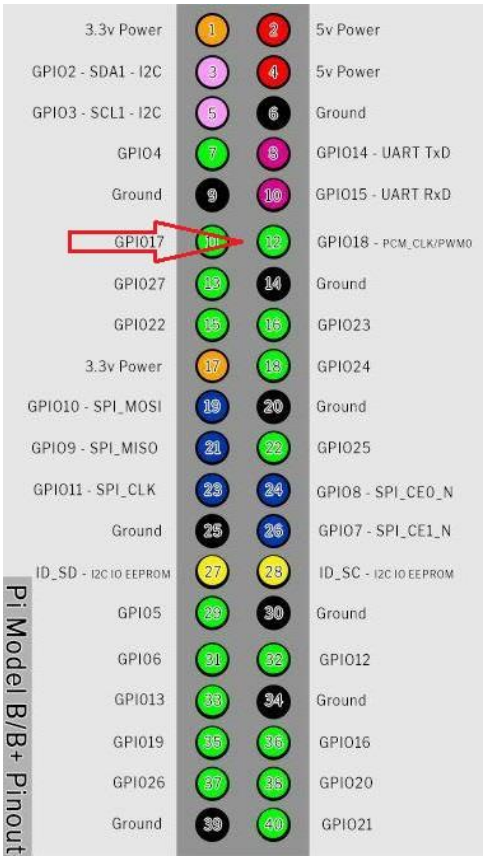


Рисунок 8.8 – Діаграма GPIO пінів Raspberry Pi 3 B, номер піну “PWM0”

Фізичне з'єднання сервоприводу із мікрокомп'ютером було проведене з допомогою так званих “jumper” дротів типу male-female. Для цього було використано три таких дроти.

Рисунок 8.9– “Jumper” дроти типу male-female

8.3 Проектування прототипу транспортного засобу

Для того, щоб наочно продемонструвати можливості системи було вирішено спроектувати пристрій, який використовував би сервопривод у якості механізму дверей, імітуючи двері транспортного засобу. Для цього було обрано просту невелику шухляду.

Рисунок 8.10 – Шухляда-імітатор транспортного засобу

Для того, щоб можно було під'єднати прототип до джерела живлення, у шухляді було зроблено невеликий отвір.

Сервопривод SG90 було розміщено на дверцятах шухляди, а у стіні шухляди, розміщеної перпендикулярно до дверцят зроблено виїм для насадки приводу. Таким чином імітується зачинення/відчинення замка транспорту.

8.4 Висновки

В результаті роботи над прототипом, було створено демонстраційний екземпляр системи цифрових ключів QKey, який надає користувачеві можливість оцінити функціонал системи на базовому рівні та пересвідчитись у її перевагах.

ВИСНОВКИ

Проаналізувавши предметну галузь та існуючі рішення, було прийнято рішення про створення системи цифрових ключів QKey, яка суміщатиме в собі найкраще з усіх представлених на ринку продуктів, нівелюючи недоліки кожного з них наскільки це можливо.

В ході проектування системи, було виділено три основних компонента:

- **Firebase backend** – серверна частина;
- **Android application** – мобільний додаток для кінцевого користувача;
- **IoT application** – додаток транспортного засобу.

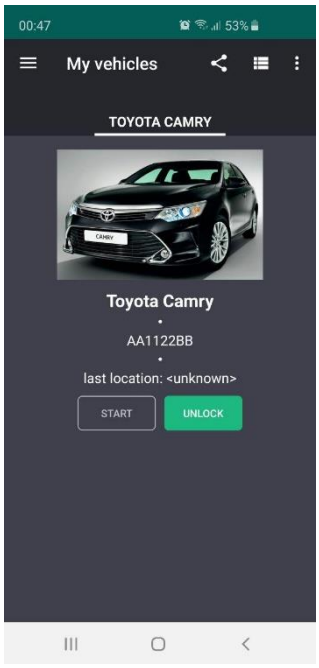
Центральним компонентом системи QKey є Firebase сервер, адже він зберігає усі дані користувачів. Саме тому, на початку роботи над реалізацією даного продукту, було розроблено модуль, який надаватиме додаткам спільний, універсальний, простий у користуванні метод доступу до серверу, оминаючи тонкощі реалізаціїї Firebase SDK.

Після налаштування серверної частини, було розроблено додаток для операційної системи Android, який надає юзеру увесь необхідний функціонал для базового знайомства із системою.

На останньому етапі розробки ПЗ, було створено IoT додаток для транспортного засобу, який максимально просто і водночас ефективно демонструє його можливості на ряду із можливостями системи.

Остаточним, завершальним кроком роботи над проектом була розробка прототипу транспортного засобу для максимальної реалістичності імітації функціоналу системи.

На Рисунках 9.1-9.2 та 9.3-9.4 зображено відповідно результати роботи системи для власника транспортного засобу та його друга.



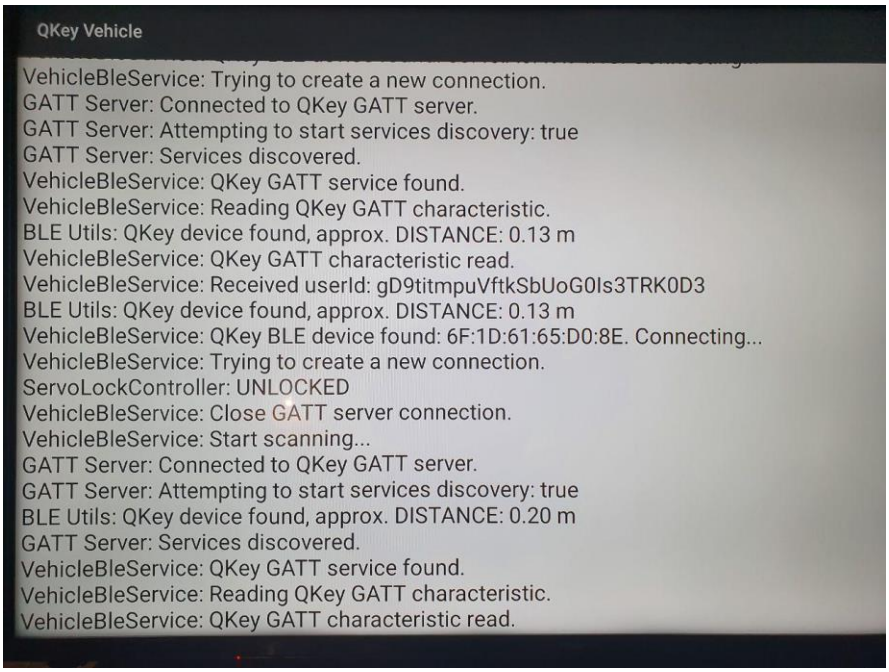
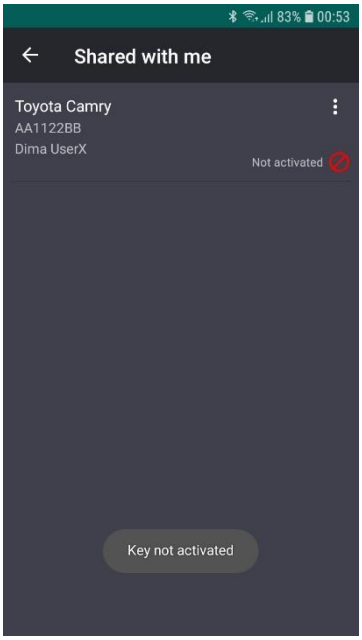


Рисунок 9.1-9.2 – екран головної активності додатку смартфона QKey та стан додатку QKey Vehicle після взаємодії відповідно (власник)

Як видно з Рисунків 9.1-9.2 користувач пробує відчинити транспортний засіб Toyota Camry, власником якого він є, підносячи смартфон на оціночну відстань 13 см і додаток QKey Vehicle миттєво реагує відчиняючи транспорт.



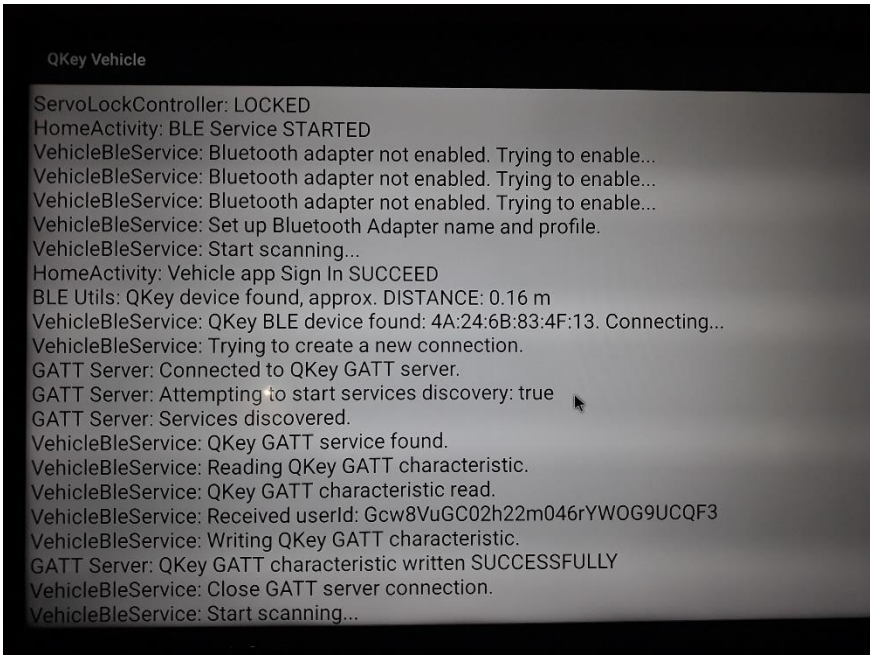


Рисунок 9.3-9.4 – екран головної активності додатку смартфона QKey та стан додатку QKey Vehicle після взаємодії відповідно (друг)

Аналогічна ситуація зображена на Рисунках 9.3-9.4. Проте в даному випадку, власник смартфону не є власником транспортного засобу і йому було відмовлено у доступі, адже його цифровий ключ наразі не активований.

З огляду на усе вище описане, я вважаю, що мета даного проекту була цілком успішно досягнута.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. https://developer.android.com/things
2. https://www.hackster.io/brandmooffin/android-things-remote-led-74ab28
3. Raspberry Pi 3 B User manual – 10 c
4. https://iotandelectronics.wordpress.com/2016/10/07/how-to-calculate-distance-from-the-rssi-value-of-the-ble-beacon/
5. https://github.com/androidthings/drivers-samples/tree/master/pwmservo
6. https://www.youtube.com/watch?v=llHi5qYqM2Y
7. Методичні вказівки до виконання курсових проектів для студентів спеціальності «Системна інженерія»
8. https://www.hackster.io/brandmooffin/android-things-bluetooth-communication-5ad434
9. https://developer.android.com/guide/topics/connectivity/bluetooth-le
10. https://developer.android.com/guide/components/services
11. https://firebase.google.com/docs/android/setup?hl=RU
12. https://firebase.google.com/docs/database/android/start?hl=RU
13. https://firebase.google.com/docs/storage/android/start?hl=RU
14. https://firebase.google.com/docs/functions/get-started?hl=RU
15. https://firebase.google.com/docs/auth/android/start?hl=RU

Додаток А

Firebase module

А.1. AuthManagerImpl – допоміжний класс для роботи з сервісом аутентифікації Firebase

```
package com.quantag.firebase.auth

import android.app.Activity
import android.util.Log
import com.google.firebase.auth.FirebaseAuth
import com.quantag.core.utils.Res
import io.reactivex.Observable
import io.reactivex.subjects.BehaviorSubject
import java.lang.IllegalStateException

class AuthManagerImpl : AuthManager {
    private val TAG: String = AuthManager::class.java.simpleName

    private val firebaseAuth: FirebaseAuth = FirebaseAuth.getInstance()

    override fun signUp(executor: Activity, email: String, password: String):
Observable<Res<String>> {
        val observableResult = BehaviorSubject.create<Res<String>>()
        firebaseAuth.createUserWithEmailAndPassword(email, password)
            .addOnCompleteListener(executor) { task ->
                if (task.isSuccessful) {
                    Log.d(TAG, "Sign Up with email/password SUCCEED!")
                    val user = firebaseAuth.currentUser
                    observableResult.onNext(Res(true, user?.uid))
                } else {
                    Log.w(TAG, "Sign Up with email/password FAILED!",
task.exception)
                    observableResult.onNext(Res(false, null,
task.exception?.message ?: ""))
                }
            }
        return observableResult
    }

    override fun isUserSignedIn(): Boolean {
        val currentUser = firebaseAuth.currentUser
        return currentUser != null
    }

    override fun signIn(executor: Activity, email: String, password: String):
Observable<Res<String>> {
        val observableResult = BehaviorSubject.create<Res<String>>()
        firebaseAuth.signInWithEmailAndPassword(email, password)
            .addOnCompleteListener(executor) { task ->
                if (task.isSuccessful) {
                    Log.d(TAG, "Sign In with email/password SUCCEED!")
                    val user = firebaseAuth.currentUser
                    observableResult.onNext(Res(true, user?.uid))
                } else {
                    Log.w(TAG, "Sign In with email/password FAILED!",
task.exception)
                    observableResult.onNext(Res(false, null,
task.exception?.message ?: ""))
                }
```

```
            }
        return observableResult
    }

    override fun userId(): String {
        if (firebaseAuth.currentUser == null){
            throw IllegalStateException("User is not signed in!!!")
        }
        return firebaseAuth.currentUser!!.uid
    }

    override fun userEmail(): String? {
        if (firebaseAuth.currentUser == null){
            throw IllegalStateException("User is not signed in!!!")
        }
        return firebaseAuth.currentUser!!.email
    }

    override fun isUserEmailVerified(): Boolean {
        if (firebaseAuth.currentUser == null){
            throw IllegalStateException("User is not signed in!!!")
        }
        return firebaseAuth.currentUser!!.isEmailVerified
    }

    override fun checkUserEmailVerification(executor: Activity):
Observable<Res<Unit>> {
        val observableResult = BehaviorSubject.create<Res<Unit>>()
        val currentUser = firebaseAuth.currentUser!!
        currentUser.reload().addOnCompleteListener(executor) { task ->
            if (task.isSuccessful) {
                observableResult.onNext(Res(currentUser.isEmailVerified))
            } else {
                val exceptionMsg = task.exception?.message
                observableResult.onNext(Res(false, null, exceptionMsg ?: ""))
            }
        }
        return observableResult
    }

    override fun sendVerificationEmail(executor: Activity): Observable<Res<Unit>>
{
        val observableResult = BehaviorSubject.create<Res<Unit>>()
        val currentUser = firebaseAuth.currentUser!!
        currentUser.sendEmailVerification().addOnCompleteListener(executor) { task
->
            if (task.isSuccessful) {
                observableResult.onNext(Res(true))
            } else {
                val exceptionMsg = task.exception?.message
                observableResult.onNext(Res(false, null, exceptionMsg ?: ""))
            }
        }
        return observableResult
    }

    override fun sendResetPasswordEmail(email: String): Observable<Res<Unit>> {
        val observableResult = BehaviorSubject.create<Res<Unit>>()
        firebaseAuth.sendPasswordResetEmail(email).addOnCompleteListener { task ->
            if (task.isSuccessful) {
                observableResult.onNext(Res(true))
            } else {
```

```
                val exceptionMsg = task.exception?.message
                observableResult.onNext(Res(false, null, exceptionMsg ?: ""))
            }
        }
        return observableResult
    }

    override fun signOut() {
        firebaseAuth.signOut()
    }
}
```

A.2. BaseHelper – базова реалізація хелперу для роботи роботи із даними певної моделі Firebase Realtime Database

```
package com.quantag.firebase.db.helper.base

import android.util.Log
import com.google.firebase.database.*
import com.quantag.firebase.model.entity.BaseFirebaseEntity
import com.quantag.core.utils.Res
import com.quantag.core.utils.SEARCH_CHAR
import io.reactivex.Observable
import io.reactivex.subjects.BehaviorSubject
import io.reactivex.subjects.ReplaySubject

abstract class BaseHelper<T: BaseFirebaseEntity>(
    protected val databaseRef: DatabaseReference,
    private val _class: Class<T>
) {
    private val TAG = BaseHelper::class.java.simpleName

    protected fun push(): Observable<Res<String>> {
        val observableResult = ReplaySubject.create<Res<String>>()
        val id = databaseRef.push().key
        observableResult.onNext(Res(!id.isNullOrEmpty(), id));
observableResult.onComplete()
        return observableResult
    }

    protected fun create(model: T): Observable<Res<Unit>> {
        val observableResult = BehaviorSubject.create<Res<Unit>>()
        databaseRef.child(model.id).setValue(model)
            .addOnSuccessListener{ observableResult.onNext(Res(true));
observableResult.onComplete() }
            .addOnFailureListener{ observableResult.onNext(Res(false, null,
it.message?:"")); observableResult.onComplete() }
        return observableResult
    }

    protected fun update(id: String, updates: Map<String, Any?>):
Observable<Res<Unit>> {
        val observableResult = BehaviorSubject.create<Res<Unit>>()
        databaseRef.child(id).updateChildren(updates)
            .addOnSuccessListener{ observableResult.onNext(Res(true));
observableResult.onComplete() }
            .addOnFailureListener{ observableResult.onNext(Res(false, null,
it.message?:"")); observableResult.onComplete() }
        return observableResult
    }
```

```
    fun delete(id: String): Observable<Res<Unit>> {
        val observableResult = BehaviorSubject.create<Res<Unit>>()
        databaseRef.child(id).removeValue()
            .addOnSuccessListener{ observableResult.onNext(Res(true));
observableResult.onComplete() }
            .addOnFailureListener{ observableResult.onNext(Res(false, null,
it.message?:"")); observableResult.onComplete() }
        return observableResult
    }

    protected fun getById(id: String): Observable<Res<T>> {
        val observableResult = BehaviorSubject.create<Res<T>>()
        val resultListener = object : ValueEventListener {
            override fun onDataChange(dataSnapshot: DataSnapshot) {
                val model = dataSnapshot.getValue(_class)
                observableResult.onNext(Res(true, model));
observableResult.onComplete()
            }

            override fun onCancelled(databaseError: DatabaseError) {
                Log.w(TAG, "Load ${_class.simpleName}: onCancelled",
databaseError.toException())
                observableResult.onNext(Res(false)); observableResult.onComplete()
            }
        }
        databaseRef.child(id).addListenerForSingleValueEvent(resultListener)
        return observableResult
    }

    protected fun getByTextField(fieldName: String, fieldValue: String):
Observable<Res<T>> {
        val observableResult = BehaviorSubject.create<Res<T>>()
        val resultListener = object : ValueEventListener {
            override fun onDataChange(dataSnapshot: DataSnapshot) {
                if (!dataSnapshot.hasChildren()) {
                    observableResult.onNext(Res(false));
observableResult.onComplete()
                    return
                }
                for (modelSnapshot in dataSnapshot.children) {
                    val targetModel = modelSnapshot.getValue(_class)
                    observableResult.onNext(Res(true, targetModel));
observableResult.onComplete()
                    break
                }
            }

            override fun onCancelled(databaseError: DatabaseError) {
                Log.w(TAG, "Load ${_class.simpleName}: onCancelled",
databaseError.toException())
                observableResult.onNext(Res(false)); observableResult.onComplete()
            }
        }
        databaseRef.orderByChild(fieldName)
            .equalTo(fieldValue)
            .limitToLast(1)
            .addListenerForSingleValueEvent(resultListener)
        return observableResult
    }

    protected fun findByTextField(fieldName: String, queryText: String):
```

```
Observable<Res<List<T>>> {
        val observableResult = BehaviorSubject.create<Res<List<T>>>()
        val modelsListener = object : ValueEventListener {
            override fun onDataChange(dataSnapshot: DataSnapshot) {
                val models = mutableListOf<T>()
                if (!dataSnapshot.hasChildren()) {
                    observableResult.onNext(Res(false, models));
observableResult.onComplete()
                    return
                }
                for (modelSnapshot in dataSnapshot.children) {
                    val model = modelSnapshot.getValue(_class)
                    if (model != null){
                        models.add(model)
                    }
                }
                observableResult.onNext(Res(true, models));
observableResult.onComplete()
            }

            override fun onCancelled(databaseError: DatabaseError) {
                Log.w(TAG, "Load ${_class.simpleName}: onCancelled",
databaseError.toException())
                observableResult.onNext(Res(false)); observableResult.onComplete()
            }
        }
        databaseRef.orderByChild(fieldName)
            .startAt(queryText)
            .endAt(queryText + SEARCH_CHAR)
            .addListenerForSingleValueEvent(modelsListener)
        return observableResult
    }
}
```

A.3. StorageHelperImpl – допоміжний класс для роботи із медіа даними

```
package com.quantag.firebase.media.helper

import android.net.Uri
import android.util.Log
import com.google.firebase.storage.StorageMetadata
import com.google.firebase.storage.StorageReference
import com.quantag.core.utils.IMAGE_JPEG_TYPE
import com.quantag.core.utils.Res
import io.reactivex.Observable
import io.reactivex.subjects.BehaviorSubject

class StorageHelperImpl(
    private val storageRef: StorageReference
): StorageHelper {

    companion object {
        const val TAG = "StorageHelperImpl"
    }

    override fun uploadMedia(id: String, uri: Uri): Observable<Res<Unit>> {
        val observableResult = BehaviorSubject.create<Res<Unit>>()
        // Create the file metadata
        val metadata = StorageMetadata.Builder()
            .setContentType(IMAGE_JPEG_TYPE)
            .build()
```

```
        // Upload image and metadata
        val uploadTask = storageRef.child(id).putFile(uri, metadata)
        // Listen for state changes, errors, and completion of the upload.
        uploadTask.addOnProgressListener{ taskSnapshot ->
            val progress = (100.0 * taskSnapshot.bytesTransferred) /
taskSnapshot.totalByteCount
            Log.d(TAG, "Upload is $progress% done")
        }.addOnPausedListener { Log.d(TAG, "Upload is paused") }
            .addOnFailureListener { observableResult.onNext(Res(false, null,
it.message!!)) }
            .addOnSuccessListener { observableResult.onNext(Res(true)) }
        return observableResult
    }

    override fun loadMediaFileUrl(id: String): Observable<Res<Uri?>> {
        val observableResult = BehaviorSubject.create<Res<Uri?>>()
        storageRef.child(id).downloadUrl
            .addOnSuccessListener { observableResult.onNext(Res(true, it, "")) }
            .addOnFailureListener { observableResult.onNext(Res(false, null,
it.message!!)) }
        return observableResult
    }
}
```

Firebase Cloud Functions

Б.1. Key properties Cloud Functions

```
const functions = require('firebase-functions');
const admin = require('firebase-admin');
const firebaseApp = admin.initializeApp();
const db = firebaseApp.database();

// KeyProperties auto-creation
exports.userDefaultKeyProperties =
functions.database.ref('/users/{uid}').onCreate(async (snapshot) => {
  const user = snapshot.val();

  const keyPropertiesRef = await db.ref('/keyProperties/').push();
  const keyPropertiesId = keyPropertiesRef.key;
  console.log('New KeyProperties UID:', keyPropertiesId);

  try {
    await db.ref('/keyProperties/' + keyPropertiesId +
'/id').set(keyPropertiesId);
    await db.ref('/users/' + user.id +
'/defaultKeyPropertiesId').set(keyPropertiesId);
    console.log('Default KeyProperties created :', keyPropertiesId, '; user: ',
user.id);
  } catch(error) {
    console.error('There was an error while creating User default KeyProperties:',
error);
  }
  return null;
});

exports.keyCustomKeyProperties =
functions.database.ref('/keys/{uid}').onCreate(async (snapshot) => {
  const key = snapshot.val();

  const keyPropertiesRef = await db.ref('/keyProperties/').push();
  const keyPropertiesId = keyPropertiesRef.key;
  console.log('New KeyProperties UID:', keyPropertiesId);

  try {
    await db.ref('/keyProperties/' + keyPropertiesId +
'/id').set(keyPropertiesId);
    await db.ref('/keys/' + key.id + '/keyPropertiesId').set(keyPropertiesId);
    if (key.ownerId === key.userId){
      await db.ref('/vehicles/' + key.vehicleId +
'/defaultKeyPropertiesId').set(keyPropertiesId);
    }
    console.log('Default KeyProperties created :', keyPropertiesId, '; key: ',
key.id);
  } catch(error) {
    console.error('There was an error while creating Key custom KeyProperties:',
error);
  }
  return null;
});

exports.groupDefaultKeyProperties =
functions.database.ref('/shareGroups/{uid}').onCreate(async (snapshot) => {
  const shareGroup = snapshot.val();
```

```
  const keyPropertiesRef = await db.ref('/keyProperties/').push();
  const keyPropertiesId = keyPropertiesRef.key;
  console.log(`New KeyProperties UID:`, keyPropertiesId);

  try {
    await db.ref('/keyProperties/' + keyPropertiesId +
'/id').set(keyPropertiesId);
    await db.ref('/shareGroups/' + shareGroup.id +
'/defaultKeyPropertiesId').set(keyPropertiesId);
    console.log(`Default KeyProperties created :`, keyPropertiesId, `; shareGroup:
`, shareGroup.id);
  } catch(error) {
    console.error('There was an error while creating default ShareGroup
KeyProperties:', error);
  }
  return null;
});

// KeyProperties auto-deletion
exports.deleteKeyCustomProperties =
functions.database.ref('/keys/{uid}').onDelete(async (snapshot) => {
  const key = snapshot.val();
  console.log(`Dropping KeyProperties UID:`, key.keyPropertiesId);

  try {
    await db.ref('/keyProperties/' + key.keyPropertiesId).set(null);
    if (key.ownerId === key.userId) {
      await db.ref('/vehicles/' + key.vehicleId +
'/defaultKeyPropertiesId').set(null);
    }
    console.log(`Key custom KeyProperties deleted`);
  } catch(error) {
    console.error('There was an error while deleting Key custom KeyProperties:',
error);
  }
  return null;
});

exports.deleteGroupKeyProperties =
functions.database.ref('/shareGroups/{uid}').onDelete(async (snapshot) => {
  const shareGroup = snapshot.val();
  const keyPropertiesId = shareGroup.defaultKeyPropertiesId;
  console.log(`Dropping KeyProperties UID:`, keyPropertiesId);

  try {
    await db.ref('/keyProperties/' + keyPropertiesId).set(null);
    console.log(`Default ShareGroup KeyProperties deleted`);
  } catch(error) {
    console.error('There was an error while deleting default ShareGroup
KeyProperties:', error);
  }
  return null;
});
```

Б.2. Verification code Cloud Function

```
const functions = require('firebase-functions');
const admin = require('firebase-admin');
const firebaseApp = admin.initializeApp();
const nodemailer = require('nodemailer');
```

```
const gmailEmail = functions.config().gmail.email;
const gmailPassword = functions.config().gmail.password;
const mailTransport = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: gmailEmail,
    pass: gmailPassword,
  },
});

exports.sendVerificationCode = functions.database.ref('/keys/{uid}').onWrite(async
(change) => {
  const STATUS_ACTIVATED = 0;
  const snapshotBefore = change.before;
  const snapshotAfter = change.after;
  const key = snapshotAfter.val();

  var isStatusChanged = true;
  if (snapshotBefore.exists()){
    const keyBefore = snapshotBefore.val();
    if (keyBefore.status === key.status){
      isStatusChanged = false;
    }
  }
  if (!isStatusChanged) {
    return null;
  }

  const isEmailRequired = key.status < STATUS_ACTIVATED;
  if (!isEmailRequired) {
    return null;
  }

  const verificationCode = -key.status;

  const userEmailSnapshot = await firebaseApp.database().ref('/users/' +
key.ownerId + '/email').once("value");
  const userEmail = userEmailSnapshot.val();

  const vehicleNameSnapshot = await firebaseApp.database().ref('/vehicles/' +
key.vehicleId + '/name').once("value");
  const vehicleName = vehicleNameSnapshot.val();

  const vehicleNumberSnapshot = await firebaseApp.database().ref('/vehicles/' +
key.vehicleId + '/vehicleId').once("value");
  const vehicleNumber = vehicleNumberSnapshot.val();

  const mailOptions = {
    from: '"Quantag It Solutions" <noreply@firebase.com>',
    to: userEmail,
  };

  // Building Email message.
  mailOptions.subject = 'Verification code';
  mailOptions.text = 'You have shared key of ' + vehicleName +  ', ' +
vehicleNumber + '.\n'
    + 'Verification code: ' + verificationCode + '.\n';

  try {
    await mailTransport.sendMail(mailOptions);
    console.log(`New verification code email sent to:`, userEmail);
```

```
  } catch(error) {
    console.error('There was an error while sending the email:', error);
  }
  return null;
});
```

Мобільний додаток QKey

В.1. MainActivity – головна активність додатку

```
package com.quantag.qkey_user.ui.activity.main

import android.annotation.SuppressLint
import android.content.*
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.MenuItem
import androidx.core.view.GravityCompat
import androidx.navigation.Navigation
import androidx.navigation.ui.NavigationUI
import kotlinx.android.synthetic.main.activity_main.*
import org.koin.android.ext.android.inject
import org.koin.androidx.viewmodel.ext.android.viewModel
import android.net.ConnectivityManager
import android.net.Uri
import android.os.IBinder
import android.util.Log
import androidx.appcompat.widget.AppCompatTextView
import androidx.lifecycle.Observer
import com.quantag.firebase.model.entity.User
import com.quantag.qkey_user.R
import com.quantag.qkey_user.ui.activity.OnBackPressListener
import com.quantag.qkey_user.ui.activity.auth.AuthActivity
import
com.quantag.qkey_user.ui.activity.owner_verification.OwnerVerificationActivity
import com.quantag.qkey_user.utils.AndroidUtils
import com.quantag.qkey_user.utils.ImageViewLoader
import com.quantag.qkey_user.utils.MessageUtils
import de.hdodenhof.circleimageview.CircleImageView

class MainActivity : AppCompatActivity(), FragmentCallback {
    private val viewModel: MainViewModel by viewModel()
    private val imageViewLoader: ImageViewLoader by inject()
    private val androidUtils: AndroidUtils by inject()
    private val messageUtils: MessageUtils by inject()
    private lateinit var networkStateReceiver: BroadcastReceiver
    private lateinit var serviceConnection: ServiceConnection
    private lateinit var userDataObserver: Observer<User?>

    @SuppressLint("SetTextI18n")
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setupToolbar()
        setupNavigation()
        userDataObserver = Observer { user ->
            if (user == null) {
                signOut()
                messageUtils.showToast(getString(R.string.user_data_not_found))
                return@Observer
            }
            val headerView = navigationView.getHeaderView(0)
            val userPhotoView =
headerView.findViewById<CircleImageView>(R.id.userPhotoView)
            val userNameView =
headerView.findViewById<AppCompatTextView>(R.id.userNameView)
```

```
            val userInfoView =
headerView.findViewById<AppCompatTextView>(R.id.userInfoView)

            imageViewLoader.loadImageTo(Uri.parse(user.photoUrl),
R.drawable.ic_image_grey_72dp, userPhotoView, {})
            userNameView.text = "${user.firstName} ${user.lastName}"
            userInfoView.text = getString(R.string.user_id, user.userId)
        }

        serviceConnection = object : ServiceConnection {
            override fun onServiceConnected(name: ComponentName?, iBinder:
IBinder?) { Log.d(TAG, "UserBleService CONNECTED") }

            override fun onServiceDisconnected(name: ComponentName?) { Log.d(TAG,
"UserBleService DISCONNECTED") }
        }
        bindService(Intent(this,
com.quantag.qkey_user.ble.UserBleService::class.java), serviceConnection,
Context.BIND_AUTO_CREATE)
    }

    private fun setupToolbar() {
        networkStateReceiver = object : BroadcastReceiver() {
            override fun onReceive(context: Context, intent: Intent) {
                if (!androidUtils.isNetworkAvailable()){
                    toolbar.subtitle = getString(R.string.msg_progress,
getString(R.string.waiting_for_network))
                } else {
                    toolbar.subtitle = ""
                }
            }
        }
        val mNetworkStateFilter =
IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION)
        registerReceiver(networkStateReceiver, mNetworkStateFilter)

        if (!androidUtils.isNetworkAvailable()){
            toolbar.subtitle = getString(R.string.msg_progress,
getString(R.string.waiting_for_network))
        }
        setSupportActionBar(toolbar)
    }

    override fun onSupportNavigateUp(): Boolean {
        return NavigationUI.navigateUp(Navigation.findNavController(this,
R.id.navHostFragment), layoutDrawer)
    }

    private fun setupNavigation() {
        val navController = Navigation.findNavController(this,
R.id.navHostFragment)

        // Update action bar to reflect navigation
        NavigationUI.setupActionBarWithNavController(this, navController,
layoutDrawer)

        // Handle nav drawer item clicks
        navigationView.setNavigationItemSelectedListener { menuItem ->
            menuItem.isChecked = true
            layoutDrawer.closeDrawers()
            true
        }
```

```
        navigationView.getHeaderView(0).setOnClickListener {
            if (navController.currentDestination?.id != R.id.settingsFragment){
                navController.navigate(R.id.settingsFragment)
            }
            layoutDrawer.closeDrawers()
        }

        // Tie nav graph to items in nav drawer
        NavigationUI.setupWithNavController(navigationView, navController)
    }

    override fun onResume() {
        super.onResume()
        viewModel.loadUserData().observe(this, userDataObserver)
    }

    override fun onBackPressed() {
        if (layoutDrawer.isDrawerOpen(GravityCompat.START)) {
            layoutDrawer.closeDrawer(GravityCompat.START)
        } else {
            val currentFragment =
navHostFragment.childFragmentManager.fragments[0]
            if (currentFragment !is OnBackPressListener ||
                !(currentFragment as OnBackPressListener).onBackPressed()){
                super.onBackPressed()
            }
        }
    }

    fun onClickDrawerItem(menuItem: MenuItem){
        when (menuItem.itemId){
            R.id.mi_add_vehicle -> {
                val showOwnerVerification = Intent(this,
OwnerVerificationActivity::class.java)
                startActivity(showOwnerVerification)
                layoutDrawer.closeDrawers()
            }
            R.id.mi_sign_out -> {
                signOut()
            }
        }
    }

    override fun signOut(){
        viewModel.signOut()
        val showAuthIntent = Intent(this, AuthActivity::class.java)
        startActivity(showAuthIntent)
        finish()
    }

    override fun hideKeyboard() {
        androidUtils.hideKeyboard(currentFocus)
    }

    override fun onDestroy() {
        unregisterReceiver(networkStateReceiver)
        unbindService(serviceConnection)
        super.onDestroy()
    }

    companion object {
        const val TAG = "MainActivity"
```

```
    }
}
```

B.2. UserBleService – компонент додатку, відповідальний за розгортання QKey GATT серверу, налаштування відповідного сервісу та обробку BLE запитів

```
package com.quantag.qkey_user.ble

import android.app.Service
import android.content.Intent
import android.os.IBinder
import android.bluetooth.BluetoothDevice
import android.bluetooth.BluetoothGattServer
import android.bluetooth.BluetoothManager
import android.bluetooth.BluetoothAdapter
import android.content.Context
import android.content.IntentFilter
import android.content.BroadcastReceiver
import android.util.Log
import android.os.ParcelUuid
import android.bluetooth.le.AdvertiseData
import android.bluetooth.le.AdvertiseSettings
import android.bluetooth.le.BluetoothLeAdvertiser
import android.os.Handler
import android.os.Looper
import com.quantag.firebase.auth.AuthManager
import com.quantag.qkey_user.ble.QKeyBleProfile.SERVICE
import com.quantag.qkey_user.utils.MessageUtils
import org.koin.android.ext.android.inject

class UserBleService: Service() {
    private val authManager: AuthManager by inject()
    private val messageUtils: MessageUtils by inject()
    private lateinit var bluetoothManager: BluetoothManager
    private var bluetoothLeAdvertiser: BluetoothLeAdvertiser? = null
    private val bleAdvertiseCallback = UserBleAdvertiseCallback()
    private var bluetoothGattServer: BluetoothGattServer? = null
    private var gattServerCallback: UserGattServerCallback? = null
    private var registeredVehicle: BluetoothDevice? = null
    private val bluetoothReceiver = object : BroadcastReceiver() {
        override fun onReceive(context: Context, intent: Intent) {
            val state = intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
BluetoothAdapter.STATE_OFF)
            when (state) {
                BluetoothAdapter.STATE_ON -> {
                    startAdvertising()
                    startServer()
                }
                BluetoothAdapter.STATE_OFF -> {
                    stopServer()
                    stopAdvertising()
                }
            }
        }
    }

    override fun onBind(intent: Intent?): IBinder? {
        Log.d(TAG, "BLE service STARTED")
```

```
        bluetoothManager = getSystemService(Context.BLUETOOTH_SERVICE) as
BluetoothManager

        val filter = IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED)
        registerReceiver(bluetoothReceiver, filter)
        if (bluetoothManager.adapter.isEnabled) {
            Log.d(TAG, "Bluetooth enabled, starting services...")
            startAdvertising()
            startServer()
        }
        return null
    }

    private fun startAdvertising() {
        bluetoothLeAdvertiser = bluetoothManager.adapter.bluetoothLeAdvertiser
        if (bluetoothLeAdvertiser == null) {
            Log.w(TAG, "Failed to create advertiser")
            return
        }
        val settings = AdvertiseSettings.Builder()
            .setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_BALANCED)
            .setConnectable(true)
            .setTimeout(0)
            .setTxPowerLevel(AdvertiseSettings.ADVERTISE_TX_POWER_MEDIUM)
            .build()
        val data = AdvertiseData.Builder()
            .setIncludeDeviceName(true)
            .setIncludeTxPowerLevel(false)
            .addServiceUuid(ParcelUuid(SERVICE))
            .build()
        bluetoothLeAdvertiser!!.startAdvertising(settings, data,
bleAdvertiseCallback)
    }

    private fun stopAdvertising() {
        if (bluetoothLeAdvertiser == null) return
        bluetoothLeAdvertiser!!.stopAdvertising(bleAdvertiseCallback)
    }

    private fun startServer() {
        gattServerCallback = UserGattServerCallback(object :
            UserGattServerCallback.VehicleConnectionCallback {
            override fun onConnected(vehicle: BluetoothDevice) { registeredVehicle
= vehicle }

            override fun onMessage(message: String) {
                val uiHandler = Handler(Looper.getMainLooper())
                uiHandler.post { messageUtils.showToast(message) }
            }

            override fun onDisconnected() { registeredVehicle = null }
        })
        bluetoothGattServer = bluetoothManager.openGattServer(this,
gattServerCallback)
        if (bluetoothGattServer == null) {
            Log.w(TAG, "Unable to create GATT server")
            return
        }
        bluetoothGattServer!!.run {
            gattServerCallback!!.gattServer = this
            addService(QKeyBleProfile.createQKeyBleService(authManager.userId()))
        }
```

```
    }

    private fun stopServer() {
        gattServerCallback = null
        if (bluetoothGattServer == null) return
        bluetoothGattServer!!.close()
        bluetoothGattServer = null
    }

    override fun onUnbind(intent: Intent?): Boolean {
        Log.d(TAG, "BLE service CLOSED")
        val bluetoothAdapter = bluetoothManager.adapter
        if (bluetoothAdapter.isEnabled) {
            stopServer()
            stopAdvertising()
        }
        unregisterReceiver(bluetoothReceiver)
        return super.onUnbind(intent)
    }

    companion object {
        const val TAG = "UserBleService"
    }
}
```

Додаток транспортного засобу QKey Vehicle

Г.1. HomeActivity – головна активність додатку

```
package com.quantag.qkey_vehicle.ui

import android.annotation.SuppressLint
import android.app.Activity
import android.content.ComponentName
import android.content.Context
import android.content.Intent
import android.content.ServiceConnection
import android.os.Bundle
import android.os.IBinder
import android.view.Menu
import com.quantag.qkey_vehicle.R
import com.quantag.qkey_vehicle.ble.VehicleBleService
import com.quantag.qkey_vehicle.ble.logger
import com.quantag.qkey_vehicle.logger.TextViewLogger
import kotlinx.android.synthetic.main.activity_home.*
import android.widget.ScrollView
import com.quantag.firebase.auth.AuthManager
import com.quantag.qkey_vehicle.lock.LockController
import com.quantag.qkey_vehicle.lock.RaspSG90ServoPWMDriver
import com.quantag.qkey_vehicle.utils.VehicleUtils
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers
import org.koin.android.ext.android.inject

class HomeActivity: Activity() {
    private lateinit var bleServiceConnection: ServiceConnection
    private val authManager: AuthManager by inject()
    private val vehicleUtils: VehicleUtils by inject()
    private val lockDriver: RaspSG90ServoPWMDriver by inject()
    private val lockController: LockController by inject()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_home)
        initUI()
        initFirebase()
        initLock()
        initBLE()
    }

    override fun onCreateOptionsMenu(menu: Menu?): Boolean {
        menuInflater.inflate(R.menu.menu_home, menu)
        menu!!.findItem(R.id.mi_close).setOnMenuItemClickListener { finish(); true
}
        return super.onCreateOptionsMenu(menu)
    }

    private fun initUI() {
        logger = TextViewLogger(logView)
        logView.addOnLayoutChangeListener { _, _, _, _, bottom, _, _, _, bottomWas
-> if (bottom > bottomWas) {
            layoutRoot.post { layoutRoot.fullScroll(ScrollView.FOCUS_DOWN) }
        }}
    }
```

```
    @SuppressLint("CheckResult")
    private fun initFirebase() {
        authManager.signIn(this, vehicleUtils.getVehicleFirebaseEmail(),
vehicleUtils.getVehicleFirebasePassword())
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(
                { logger.log(TAG, "Vehicle app Sign In SUCCEED") },
                { logger.log(TAG, it.message ?: "Vehicle app Sign In FAILED");
finish() }
            )
    }

    private fun initBLE() {
        val bleServiceIntent = Intent(this, VehicleBleService::class.java)
        bleServiceConnection = object : ServiceConnection {
            override fun onServiceConnected(name: ComponentName?, iBinder:
IBinder?) {
                logger.log(TAG, "BLE Service STARTED")
                val bleService = (iBinder as
VehicleBleService.LocalBinder).service
                Thread().run { while (!bleService.initialize()) {
                    Thread.sleep(2000)
                    continue
                }}
            }

            override fun onServiceDisconnected(name: ComponentName?) {
                logger.log(TAG, "BLE Service CLOSED")
            }
        }
        bindService(bleServiceIntent, bleServiceConnection,
Context.BIND_AUTO_CREATE)
    }

    private fun initLock() { lockController.lock() }

    override fun onDestroy() {
        unbindService(bleServiceConnection)
        lockDriver.clear()
        super.onDestroy()
    }

    companion object {
        const val TAG = "HomeActivity"
    }
}
```

Г.2. VehicleBleService – компонент додатку, відповідальний за сканування ефіру на предмет QKey GATT серверів за подальшу обробку з’єднання BLE пристроїв

```
package com.quantag.qkey_vehicle.ble

import android.app.Service
import android.bluetooth.*
import android.content.Context
import android.content.Intent
import android.os.Binder
```

```
import android.os.IBinder
import com.quantag.core.utils.DEFAULT_ERROR
import com.quantag.core.utils.QKEY_BLE_PROFILE_DATA_UUID
import com.quantag.core.utils.QKEY_BLE_PROFILE_MESSAGE_UUID
import com.quantag.core.utils.QKEY_BLE_PROFILE_SERVICE_UUID
import
com.quantag.qkey_vehicle.ble.VehicleGattServerCallback.ServerConnectionCallback
import com.quantag.qkey_vehicle.key_manager.DigitalKeyManager
import com.quantag.qkey_vehicle.lock.LockController
import io.reactivex.disposables.CompositeDisposable
import org.koin.android.ext.android.inject
import java.util.*

class VehicleBleService: Service() {
    private val binder = LocalBinder()
    private var bluetoothManager: BluetoothManager? = null
    private var bluetoothAdapter: BluetoothAdapter? = null
    private var qkeyGatt: BluetoothGatt? = null
    private var qkeyGattService: BluetoothGattService? = null
    private lateinit var vehicleGattServerCallback: VehicleGattServerCallback
    private var serverConnectionCallback: ServerConnectionCallback? = null
    private lateinit var messageSentListener: () -> Unit
    private var bluetoothDeviceAddress: String? = null
    private val lockController: LockController by inject()
    private val digitalKeyManager: DigitalKeyManager by inject()
    private val compositeSubscription = CompositeDisposable()

    override fun onBind(intent: Intent): IBinder? = binder

    fun initialize(): Boolean {
        if (bluetoothManager == null) {
            bluetoothManager = getSystemService(Context.BLUETOOTH_SERVICE) as
BluetoothManager
            if (bluetoothManager == null) {
                logger.log(TAG, "Unable to initialize BluetoothManager.")
                return false
            }
        }
        bluetoothAdapter = bluetoothManager!!.adapter
        if (bluetoothAdapter == null) {
            logger.log(TAG, "Unable to obtain a BluetoothAdapter.")
            return false
        }
        if (!bluetoothAdapter!!.isEnabled) {
            logger.log(TAG, "Bluetooth adapter not enabled. Trying to enable...")
            bluetoothAdapter!!.enable()
            return false
        }
        logger.log(TAG, "Set up Bluetooth Adapter name and profile.")
        bluetoothAdapter!!.name = ADAPTER_FRIENDLY_NAME
        startScan()
        return true
    }

    private fun startScan() {
        logger.log(TAG, "Start scanning...")
        ScanTask(bluetoothAdapter!!.bluetoothLeScanner) { deviceAddress ->
            logger.log(TAG, "QKey BLE device found: $deviceAddress.
Connecting...")
            connectToDevice(deviceAddress)
        }.execute()
    }
```

```
    private fun connectToDevice(address: String): Boolean {
        // Previously connected device.  Try to reconnect.
        if (bluetoothDeviceAddress != null && address == bluetoothDeviceAddress &&
qkeyGatt != null) {
            logger.log(TAG, "Trying to use an existing qkeyGatt for
reconnection.")
            return qkeyGatt!!.connect()
        }
        val device = bluetoothAdapter!!.getRemoteDevice(address)
        if (device == null) {
            logger.log(TAG, "Device not found.  Unable to connect to device
$address.")
            return false
        }
        logger.log(TAG, "Trying to create a new connection.")
        vehicleGattServerCallback =
VehicleGattServerCallback(buildServerConnectionCallback())
        qkeyGatt = device.connectGatt(this, false, vehicleGattServerCallback)
        bluetoothDeviceAddress = address
        return true
    }

    private fun buildServerConnectionCallback(): ServerConnectionCallback {
        if (serverConnectionCallback == null) {
            serverConnectionCallback = object : ServerConnectionCallback {
                override fun onServicesDiscovered() {
                    val services = if (qkeyGatt == null) null else
qkeyGatt!!.services
                    if (!services.isNullOrEmpty()) {
                        qkeyGattService = services.find { it.uuid ==
QKEY_BLE_SERVICE }
                        logger.log(TAG, "QKey GATT service found.")
                        if (qkeyGattService != null) {
                            val qkeyCharacteristic =
qkeyGattService!!.getCharacteristic(QKEY_BLE_DATA)
                            logger.log(TAG, "Reading QKey GATT characteristic.")
                            if (qkeyCharacteristic != null) {
qkeyGatt!!.readCharacteristic(qkeyCharacteristic) }
                        }
                    }
                }

                override fun onDataAvailable(gattData:
BluetoothGattCharacteristic) {
                    logger.log(TAG, "QKey GATT characteristic read.")
                    handleGattData(gattData)
                }

                override fun onMessageSent() { messageSentListener.invoke() }

                override fun onDisconnected() { startScan() }
            }
        }
        return serverConnectionCallback!!
    }

    private fun handleGattData(gattData: BluetoothGattCharacteristic) {
        val userId = String(gattData.value)
        logger.log(TAG, "Received userId: $userId")
        val performReconnection = { close(); startScan() }
        val subscription = digitalKeyManager.checkUserAccess(userId)
```

```
            .subscribe(
                {
                    if (it.success) {
                        lockController.toggleLock()
                        digitalKeyManager.onKeyUsed()
                        performReconnection.invoke()
                    } else sendMessageToServer(it.errorMsg) {
performReconnection.invoke() }
                },
                {
                    it.message?.run { logger.log(TAG, this) }
                    performReconnection.invoke()
                }
            )
        compositeSubscription.add(subscription)
    }

    private fun sendMessageToServer(message: String, sentListener: () -> Unit) {
        messageSentListener = sentListener
        val messageCharacteristic =
qkeyGattService!!.getCharacteristic(QKEY_BLE_MESSAGE)
        messageCharacteristic.setValue(message)
        logger.log(TAG, "Writing QKey GATT characteristic.")
        qkeyGatt?.writeCharacteristic(messageCharacteristic)
    }

    override fun onDestroy() {
        close()
        compositeSubscription.dispose()
        super.onDestroy()
    }

    private fun close() {
        logger.log(TAG, "Close GATT server connection.")
        if (qkeyGatt == null) { return }
        qkeyGatt!!.apply {
            disconnect()
            close()
        }
        qkeyGatt = null
    }

    inner class LocalBinder : Binder() {
        internal val service: VehicleBleService
            get() = this@VehicleBleService
    }

    companion object {
        const val TAG = "VehicleBleService"
        const val ADAPTER_FRIENDLY_NAME = "Raspberry Pi 3 B"

        val QKEY_BLE_SERVICE = UUID.fromString(QKEY_BLE_PROFILE_SERVICE_UUID)
        val QKEY_BLE_DATA = UUID.fromString(QKEY_BLE_PROFILE_DATA_UUID)
        val QKEY_BLE_MESSAGE = UUID.fromString(QKEY_BLE_PROFILE_MESSAGE_UUID)
    }
}
```

Г.3. ServoLockController – контроллер для керування сервоприводом

```
package com.quantag.qkey_vehicle.lock
```

```
import com.quantag.qkey_vehicle.ble.logger

class ServoLockController(
    private val pwmServoDriver: RaspSG90ServoPWMDriver
): LockController {
    companion object {
        const val TAG = "ServoLockController"
        const val LOCKED_SERVO_ANGLE = 70.0
        const val UNLOCKED_SERVO_ANGLE = 0.0
    }

    override fun toggleLock() { if (isLocked()) unlock() else lock() }

    private fun isLocked(): Boolean = pwmServoDriver.currentAngle ==
LOCKED_SERVO_ANGLE

    override fun lock() {
        pwmServoDriver.rotate(LOCKED_SERVO_ANGLE)
        logger.log(TAG, "LOCKED")
    }

    override fun unlock() {
        pwmServoDriver.rotate(UNLOCKED_SERVO_ANGLE)
        logger.log(TAG, "UNLOCKED")
    }
}
```